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

3 de abril 2018

Descripción

En su mayoría, los lenguajes de programación hacen uso de funciones[1] que facilitan la solución de problemas. Sin embargo, no funcionan de la misma manera en todos lenguajes de programación. En esta nota se revisan los tipos de funciones que existen y los retos que trae consigo su implementación.

Índice general

3.1 El lenguaje FWAE **3**

Clasificación de funciones 4

Análisis sintáctico de `FWAE` 4

3.2 Azúcar sintáctica **5**

3.3 Análisis semántico de FWAE **7**

Ambientes de evaluación 9

3.4 Alcance **11**

3.4 Referencias **13**

[1] procedimientos, subrutinas, métodos, reglas, etcétera.

Índice de códigos

Código 1: Uso de funciones en Haskell 4
Código 2: Tipo de dato abstracto `FWAE` 5
Código 3: Función `parse` que realiza el análisis sintáctico de `FWAE` 5
Código 4: Tipo de dato abstracto `FAE` 6
Código 5: Función `desugar` que elimina el azúcar sintáctica de `FWAE` 6
Código 6: Función `interp` (versión 1) 7
Código 7: Función `subst` 7
Código 8: Tipo de dato abstracto `Env` 9
Código 9: Función `lookup` 10
Código 10: Función `interp` (versión 2) 10
Código 11: TDA `FAE-Value` para representar las cerraduras 12
Código 12: Función `interp` (versión 3) 12
Código 13: Tipo de dato abstracto `Env` (versión 2) 13

Índice de figuras

Figura 1: Gramática del lenguaje `FWAE` 3
Figura 2: Ambiente de evaluación de forma gráfica 10

3.1 El lenguaje FWAE

En estas notas se extiende el lenguaje `WAE` que se estudió en la nota anterior. Se agregan 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 3.1 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, se tienen algunas diferencias en cuanto a su comportamiento, dependiendo del lenguaje de programación elegido. 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 (`duplica`) como parámetro. Haskell a su vez provee mecanismos para regresar funciones como resultado, e incluso almacenarlas como valor en variables o estructuras de datos. En otros lenguajes como C esto no es posible, ya que las funciones sólo pueden aplicarse a valores.

En general, las funciones de un lenguaje de programación, se implementan de acuerdo a la siguiente clasificación[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 construyen a lo largo de esta nota, usan funciones de primera clase.

Análisis sintáctico de FWAE

Para construir el Árbol de Sintaxis Abstracta (ASA) de 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?) (value FWAE?) (body FWAE?)]
   [funS   (param symbol?) (body FWAE?)]
   [appS   (fun-expr FWAE?) (arg FWAE?)])
```

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

El análisis sintáctico se realiza de forma parecida al descrito en la nota anterior 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 value) body)
         (withS id (parse value) (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`

3.2 Azúcar sintáctica

Antes de realizar la etapa de análisis semántico, algunos intérpretes y compiladores realizan una etapa intermedia de transformación que consiste en quitar el azúcar sintáctica de las expresiones del lenguaje[2]. 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, la siguiente expresión:

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

Es una versión simplificada que facilita la escritura de:

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

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}
```

Las asignaciones locales, son en realidad aplicaciones de función endulzadas. Se necesita, 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-expr 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 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 value body)
         (app (fun id (desugar body)) (desugar value))]
      [(funS param body) (fun param (desugar body))]
      [(appS fun-expr arg) (app (desugar fun) (desugar arg))]))
```

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

3.3 Análisis semántico de FWAE

El análisis semántico se realiza sobre las expresiones de tipo `FAE`, lo cual simplifica su implementación al no tener 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 expresiones de tipo `FAE` para evitar futuros 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-expr arg)
      (interp
        (subst
          (fun-body fun-expr)
          (fun-param fun-expr)
          (interp arg)))]))
```

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

El procedimiento `subst` requiere de algunas modificaciones para procesar aplicaciones de función. 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-expr arg)
     (app (subst fun-expr sub-id val) (subst arg sub-id val))]))
```

Código 7: *Función* `subst`

Ejemplo 3.2 Se muestra el resultado de cada etapa sobre una expresión para mostrar los distintos tipos de análisis que se han implementado hasta este punto. 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) desendulzado.

```
(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 3.2 se aprecia cómo por cada expresión `with` que aparece se hace una llamada al algoritmo de sustitución (en este caso 3 veces). Si el programa tuviera un tamaño n^2, entonces cada sustitución recorre el resto del programa al menos una vez, haciendo que su complejidad se vuelva cuadrática, es decir, $O(n^2)$.

Para cambiar este orden, se propone el uso de *ambientes* de evaluación implementados con alguna estructura de datos en la cual las búsquedas sean de un orden menor. Para los fines de estas notas, se usan un ambientes con comportamiento de pila pues la complejidad de realizar una búsqueda es lineal $O(n)$. Para representar ambientes se define en el Código 8 el TDA `Env`.

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

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

La sustitución de identificadores se realiza buscando el valor correspondiente en el ambiente. De forma gráfica se puede pensar en el ambiente como una tabla donde la primera columna almacena los identificadores correspondientes y la segunda el valor asociado.

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

Figura 2: *Ambiente de evaluación de forma gráfica*

En la Figura 2 se observa la representación gráfica del siguiente código:

[2]El número de nodos del árbol de sintaxis abstracta

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

Los identificadores ingresan a la pila conforme se va leyendo la expresión, es decir, como aparecen en el código del programador.

Para buscar un identificador en el ambiente, se presenta el Código 9, éste muestra la implementación de una función `lookup` que compara el valor a sustituir con cada identificador en el ambiente. La búsqueda se realiza de forma recursiva.

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

Código 9: *Función* `lookup`

Una 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-expr arg)
         (let ([fun-val (interp fun-expr 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 interpretan el cuerpo de la función correspondiente en el ambiente que se construye ingresando el parámetro formal de la función asociado al parámetro real que recibe la aplicación de función. De esta forma, la expresión del Ejemplo 4.2[3]:

[3]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` debe tomar la función `{fun {y} {+ x y}}` `(num 5)` o `(num 3)`?

3.4 Alcance

Después de que el lector ejecute el programa anterior, se observa que el intérprete regresa el valor `(num 9)` siendo que en el Ejemplo 3.2 se obtuvo el valor `(num 7)` ¿cuál es entonces el resultado correcto?

Esto tiene que ver con el concepto de *alcance*. De manera intuitiva, en la expresión

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

la `{fun {y} {+ x y}}` toma el valor `{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]. Estos dos casos corresponden con dos tipos de alcance[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.*

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 toma valores revisando todo el programa desde el inicio, donde se tiene que `x = 3`, por lo tanto `{f 4} = {+ 3 4} = 7`.

Ambos tipos de alcance son correctos y dependen de la visión que tiene el diseñador o los diseñadores del lenguaje de programación, al elegir alguno de estos, sin embargo, es más usado el alcance estático al ser lo que la intuición del programador espera.

La primera versión del analizador semántico implementa alcance estático pues realiza la sustitución antes que otra cosa se ejecute, por ejemplo, la subexpresión

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

se sustituye al inicio por

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

Se necesita, que el intérprete basado en ambientes, no evalúe funciones a sí mismas, y que por el contrario, cargue el ambiente donde éstas fueron definidas. Este tipo de estructura recibe se conoce como *cerradura*[4] y almacena[2]:

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

Para representar 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*

Al evaluar aplicaciones de función, el valor de los identificadores se busca 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)]
```

[4] *Closure.*

```
        [(app fun-expr 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 (id symbol?) (value FAE-Value?) (env Env?)])
```

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

3.5 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 del curso 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 4: Lenguajes de Programación`.