Lenguajes de Programación, 2019-2
Nota de laboratorio 6
Funciones de primera clase

Manuel Soto Romero

11 de marzo de 2019

Facultad de Ciencias, UNAM

El estilo de programación funcional tiene como principal unidad el uso de funciones. Las funciones de este paradigma son *miembros de primera clase*, esto quiere decir que pueden pasarse como parámetro a otras funciones, devolverse como resultado e incluso almacenarse en estructuras de datos. En esta nota se revisa una segunda versión del intérprete para el lenguaje `LDP` definido en las notas de laboratorio anteriores de manera que procese funciones de primera clase.

1. Gramática para funciones

La gramática en EBNF para el lenguaje `LDP` incluyendo funciones es la siguiente:

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

<id> ::= A | B | ... | a | ... | z | aa | ...
        (Cualquier combinación de caracteres alfanumérica)

<num> ::= ... | -1 | 0 | 1 | ... | 2.5
          (Cualquier número entero o flotante)

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

Observación 1. Recordar que `LDP` es un lenguaje de programación muy parecido sintáctica y semánticamente a Racket, el metalenguaje que se usará a lo largo de estas notas. Para diferenciar entre un lenguaje y otro se hace uso de llaves en lugar de corchetes y se usa el símbolo (λ) como apuntador (*prompt*) de `LDP`.

1.1. Tipos de datos básicos

El único tipo de dato de `LDP` son los números y son admitidos números enteros, flotantes, racionales y complejos. Además, los números pueden ser de cualquier longitud.

```
(λ) -1
-1
(λ) 1729
1729
(λ) 18.35
18.35
(λ) 1/2
1/2
(λ) 5+3i
5+3i
(λ) 17283982937489234750
17283982937489234750
```

1.2. Funciones predefinidas

1.2.1. Funciones aritméticas

Para operar con números se tienen las siguientes funciones:

- La función suma (+) permite realizar sumas de forma binaria.

```
(λ) {+ 1 2}
3
(λ) {+ {+ 1 2} {+ 3 4}}
10
```

- La función resta (-) permite realizar restas de forma binaria.

```
(λ) {- 5.0 3.5}
1.5
(λ) {- {+ 1 2} {- 3 2}}
2
```

- La función producto (*) permite realizar multiplicaciones de forma binaria.

```
(λ) {* 2.5 4.0}
10.0
(λ) {* {+ 1 4} {- 4 2}}
10
```

- La función división (/) permite realizar divisiones de forma binaria.

```
(λ) {/ 10 4}
2.5
(λ) {/ {+ 8 {- 4 2}} {* 2 2}}
2.5
```

1.3. Asignaciones locales

Para poder asignar variables con un alcance restringido, en LDP se tiene la primitiva `with`.

```
(λ) {with {x 5} x}
5
(λ) {with {x 5}
       {with {y {+ x x}}
           {* x y}}}
50
```

1.4. Funciones anónimas

Para definir funciones anónimas (lambdas) se usa la primitiva `fun`, la aplicación de una función consiste en poner entre corchetes la función y el argumento con el cual se desea aplicar.

```
(λ) {fun {x} x}
#<function>
(λ) {{fun {x} x} 2}
2
(λ) {with {foo {fun {x} {+ x 2}}}
       {foo 2}}
4
```

2. Análisis sintáctico

Para representar la sintaxis abstracta, se define, en el Listado de código 1, el tipo de dato `FWAE`.

```
;; Tipo de dato para los ASA de LDP.
(define-type FWAE
   [idS    (i symbol?)]
   [numS   (n number?)]
   [binopS (f procedure?) (lhs FWAE?) (rhs FWAE?)]
   [withS  (id symbol?) (value FWAE?) (body FWAE?)]
   [funS   (param symbol?) (body FWAE?))]
   [appS   (fun-expr FWAE?) (arg FWAE?)])
```

Listado de código 1: Tipo de dato FWAE

Observación 2. Los constructores del tipo de dato `FWAE` terminan con una letra `S`, más adelante, en la Sección 3 se explica el por qué de esta nomenclatura.

En el Listado de código 2 se implementa la función `parse` que realiza el análisis sintáctico correspondiente.

```
;; Analizador sintáctico de LDP.
;; 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 izq der)
         (binopS (elige op) (parse izq) (parse der))]
      [(list fun-expr arg)
         (appS (parse fun-expr) (parse arg))]))
```

Listado de código 2: Analizador sintáctico de LDP

Se añaden dos nuevos casos a la función implementada en la Nota de Laboratorio 4:

Funciones

Las funciones se representan mediante listas (con tres elementos que siempre inician con el símbolo `'fun`) y se transforman a expresiones de tipo `FWAE` mediante el constructor `funS`. El constructor `funS`, recibe:

- Un parámetro de tipo `symbol`, se toma tal cual de la expresión recibida sin realizar ningún tipo de transformación.
- Un cuerpo de tipo `FWAE`, éste debe procesarse recursivamente para obtener una expresión del mismo tipo.

Aplicaciones de función

Las aplicaciones de función se representan mediante listas (con dos elementos) y se transforman a expresiones de tipo `FWAE` mediante el constructor `appS`. El constructor `appS`, recibe:

- Un valor asociado a la función y un argumento (parámetro real) de tipo `FWAE`, estos deben procesarse recursivamente para obtener expresiones del mismo tipo.

3. Azúcar sintáctica

Azúcar sintáctica es aquella sintaxis en un lenguaje de programación que permite leer o expresar de forma más sencilla ciertas expresiones del lenguaje, se dice que *endulza* el lenguaje para el uso humano. En el lenguaje `LDP`, las asignaciones locales `with` son azúcar sintáctica de aplicaciones de función. Esto es:

Cuando el usuario escribe:

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

En realidad se tiene una aplicación de función de la siguiente manera:

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

El identificador de `with` pasa a ser el parámetro (formal) de la función, el cuerpo del `with` pasa a ser el cuerpo de la función y finalmente el valor asociado al identificador de `with` pasa a ser el argumento (parámetro real) de la función.

A nivel programación, el contrar con expresiones que tiene azúcar sintáctica, facilita la interpretación de las mismas al disminuir el número de casos a considerar, sin embargo, debe añadirse un procedimiento extra que elimine el azúcar sintáctica de las expresiones, es decir, debe hacer una transformación de código. En el Listado de código 3 se implementa la función `desugar` que realiza esta transformación, esta función, recibe datos de tipo `FWAE` y regresa datos de tipo `FAE` (este es el por qué de la `S` en los constructores de `FWAE`).

```
1  ;; Tipo de dato para los ASA de LDP sin azúcar sintáctica.
2  (define-type FAE
3     [id    (i symbol?)]
4     [num   (n number?)]
5     [binop (f procedure?) (lhs FAE?) (rhs FAE?)]
6     [fun   (param symbol?) (body FAE?))]
7     [app   (fun-expr FAE?) (arg FAE?)])
8
9  ;; Desendulzador de LDP.
10 ;; desugar: FWAE -> FAE
11 (define (desugar expr)
```

```
12    (match expr
13       [(idS i) (id i)]
14       [(numS n) (num n)]
15       [(binopS f izq der) (binop f (desugar izq) (desugar der))]
16       [(withS id value body)
17          (app (fun id (desugar body)) (desugar value))]
18       [(funS param body)
19          (fun param (desugar body))]
20       [(appS fun-expr arg)
21          (app (desugar fun-expr) (desugar arg))]))
```

Listado de código 3: Desendulzamiento de expresiones

Las líneas 16 y 17 muestran el desendulzamiento de expresiones `with`, el resto de expresiones procesan recursivamente cada una de sus partes.

4. Análisis semántico

A continuación se revisa el comportamiento de las nuevas primitivas del lenguaje.

Funciones Las funciones simplemente se evalúan a función. El REPL de `LDP` simplemente muestra `#<function>` al evaluar una función.

```
(λ) {fun {x} x}
#<function>
```

Aplicación de funciones Para aplicar funciones, se hace uso del algoritmo de sustitución textual, se debe sustituir cada presencia del parámetro (formal) en el cuerpo de la función por el argumento (parámetro real) recibido.

```
(λ) {{fun {x} x} 5}
5
```

El Listado de código 4 muestra la implementación de la función `interp` (líneas 20 a 26) para el lenguaje `LDP` así como la actualización del algoritmo de sustitución presentado en la Nota de laboratorio 5 (líneas 3 a 16).

```
;; Algoritmo de sustitución textual.
;; 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)
                expr
                (fun param (subst body sub-id val)))]
      [(app fun-expr arg)
         (app (subst fun-expr sub-id val) (subst arg fun-expr val))]))

;; Interpretación de expresiones de WAE.
;; interp: WAE -> number
(define (interp expr)
   (match expr
      [(id i) (error 'interp "Free identifier")]
      [(num n) n]
      [(binop f izq der) (f (interp izq) (interp der))]
      [(fun param body) expr]
     [(app fun-expr arg)
         (let ([fun-val (interp fun-expr)])
            (interp (fun-body fun-val)
                    (fun-param fun-val)
                    (interp arg)))]))
```

Listado de código 4: Implementación del algoritmo de sustitución textual