UNIVERSIDAD DEL VALLE

FACULTAD DE INGENIERÍA

Proyecto final – Informe

AUTORES:

JERSSON DANIEL GUTIERREZ GONZALEZ (2060071)

NICOLAS GUTIERREZ RAMIREZ (2259515)

CRISTIAN LEONARDO ALBARRACIN (1968253)

NATALIA GOMEZ DELACRUZ (2160128)

DOCENTE:

CARLOS ANDRES DELCADO

Fundamentos de Lenguajes Programación

Tuluá, Valle del Cauca

Junio de 2024

Contenido:

La implementación de let y var

La implementación del let y el var. Lo que se hizo fue hacer una expresión que reconoce si se está creando un let o un var. En el caso del let, lo que hace es evaluarlo donde el let evalúa los argumentos que le pasan a los IDS. Luego, crea un ambiente normal extendido al cuerpo del let. Lo que está después del in. Y lo que hace el Var es lo mismo, pero él crea un modificable extend. O sea, crea un ambiente modificable. Este ambiente modificable es un vector, pero permite tener expresiones y permite tener valores. En caso de que sea una expresión, él la evalúa en el ambiente del que viene, para que las expresiones conozcan a sí mismas. Entonces, eso es lo que hace para modificarlas prácticamente, las listas. Ya que no hay ninguna función que permita modificar listas y con el vector se usa el vector-ref para modificar los vectores. Ya que con el extend-mod-env, como es un vector, permite modificar. Y cuando él hace la evaluación del var, los argumentos se conocen a sí mismos. Esto permite también la recursión.

```
;; Ligaduras modificables
(expresion ("var" (arbno identificador "=" expresion) "in" expresion) lvar-exp)
(expresion ("let" (arbno identificador "=" expresion) "in" expresion) let-exp)
```

```
(let-exp (ids rands body)
         (let ((args (eval-rands rands env)))
           (eval-expresion body
                           (extend-env ids args env))))
```

```
(lvar-exp (ids rands body)
  (let ((args (eval-rands rands (extend-mod-env ids (list->vector rands) env))))
          (eval-expresion body
                          (extend-mod-env ids (list->vector args) env))))
```

La representación de binarios, decimales y octales

Definición del Escáner
La implementación del escáner léxico se realiza mediante la especificación de una lista de reglas, cada una de las cuales describe cómo identificar ciertos tipos de tokens en el texto de entrada. La lista se denomina scanner-spec-simple-interpreter y se define como una lista de listas

Espacios en Blanco y Comentarios:

white-sp: Reconoce y omite espacios en blanco utilizando el patrón whitespace.
comment: Reconoce y omite comentarios que comienzan con % y continúan hasta el final de la línea.
Identificadores:

identificador: Reconoce identificadores que comienzan con una letra y pueden contener letras, dígitos o el carácter ?.

Números Decimales:

digitoDecimal: Reconoce números decimales positivos y negativos. Los positivos se forman con uno o más dígitos, y los negativos comienzan con un -.
Números Binarios:

digitoBinario: Reconoce números binarios que comienzan con b o -b seguido de una secuencia de ceros y unos.
Números Octales:

digitoOctal: Reconoce números octales que comienzan con 0x o -0x seguido de dígitos del 0 al 7.
Números Hexadecimales:

digitoHexadecimal: Reconoce números hexadecimales que comienzan con hx o -hx seguido de dígitos del 0 al 9 y letras de la A a la F.
Números Flotantes:
flotante: Reconoce números flotantes que contienen un punto decimal. Los positivos tienen la forma digit.digit y los negativos -digit.digit.

```
(define scanner-spec-simple-interpreter
'((white-sp
   (whitespace) skip)
  (comment
   ("%" (arbno (not #\newline))) skip)
  (identificador
   (letter (arbno (or letter digit "?"))) symbol)
  (digitoDecimal
   (digit (arbno digit)) number)
  (digitoDecimal
   ("-" digit (arbno digit)) number)
  (digitoBinario
   ("b" (or "0" "1") (arbno (or "0" "1"))) string)
  (digitoBinario
   ("-" "b" (or "0" "1") (arbno (or "0" "1"))) string)
  (digitoOctal
   ("0x" (or "0" "1" "2" "3" "4" "5" "6" "7")
   (arbno (or "0" "1" "2" "3" "4" "5" "6" "7"))) string)
  (digitoOctal
   ("-" "0x" (or "0" "1" "2" "3" "4" "5" "6" "7")
    (arbno (or "0" "1" "2" "3" "4" "5" "6" "7"))) string)
  (digitoHexadecimal
   ("hx" (or "0" "1" "2" "3" "4" "5" "6" "7" "8" "9" "A" "B" "C" "D" "E" "F")
    (arbno (or "0" "1" "2" "3" "4" "5" "6" "7" "8" "9" "A" "B" "C" "D" "E" "F"))) string)
```

```
  (digitoHexadecimal
   ("-" "hx" (or "0" "1" "2" "3" "4" "5" "6" "7" "8" "9" "A" "B" "C" "D" "E" "F")
   (arbno (or "0" "1" "2" "3" "4" "5" "6" "7" "8" "9" "A" "B" "C" "D" "E" "F"))) string)
  (flotante
   (digit (arbno digit) "." digit (arbno digit)) number)
  (flotante
   ("-" digit (arbno digit) "." digit (arbno digit)) number)
 ))
```

Cómo se implementaron while, for y switch

El switch, lo que hace es que le llegó un valor. Le llega una lista de casos en lista de expresiones y un caso por defecto. Entonces, lo que hace él es evaluar el valor y él evalúa los casos y mira si el valor que él tiene es igual a ese al valor de ese caso. Entonces, pues, si es igual, él evaluará la expresión de ese caso, evalúa lo que está a la derecha del caso, o sea, evalúa lo que se pide hacer según el caso, y pues, si ya se acabó la lista, lo que hace es ejecutar la expresión por defecto. Ya lo del while, pues es que él comprueba. Si la expresión que llega es verdadera, si es verdadera, él evalúa el cuerpo y vuelve a evaluarse el mismo, o sea, vuelve a hacer una recursión con él mismo y eso repite hasta que el bucle sea mentira o hasta que, siempre porque el usuario puede hacer recursión infinita, puede hacer un bucle infinito. Ya lo que es el for, lo que hace este for es, digamos evaluar dónde inicia, dónde termina y la suma que le va a hacer. Entonces, lo que va a hacer es que tiene un loop que evalúe si ya i llegó al final y si i llegó al final y va a ir guardando el valor numérico en ese momento. Entonces, lo que va a hacer es evaluar el cuerpo, pero donde esa variable que él dio va a tomar el valor numérico de ese momento.

```
;; Iteradores
(expresion ("for" identificador "from" expresion "until" expresion "by" expresion "do" expresion) for-exp)
(expresion ("while" expresion "{" expresion "}") while-exp)
```

```
;; Switch
(expresion ("switch" "(" expresion ")" "{"
(arbno "case" expresion ":" expresion) "default" ":" expresion "}") switch-exp)
```

```
(while-exp (boolean_exp body_exp)
    (cond
        [(eval-expresion boolean_exp env)
          (eval-expresion body_exp env)
          (eval-expresion exp env)
        ]
        [else 'void-exp]
```

```
(for-exp (var start-exp end-exp sum-exp body-exp)
  (let ((start (eval-expresion start-exp env))
        (end (eval-expresion end-exp env))
        (sum (eval-expresion sum-exp env))
       )
    (let loop ((i start))
      (when (< i end)
        (eval-expresion body-exp (extend-env (list var) (list i) env))
        (loop (+ i sum)))))
)
```

```
(switch-exp (var_exp list_caso list_exp default_exp)
  (letrec ((valor (eval-expresion var_exp env))
      (coinciden
        (lambda (caso list_e valor)
          (cond
            [(null? caso) (eval-expresion default_exp env)]
            [(equal? valor (eval-expresion (car caso) env)) (eval-expresion (car list_e) env)]
            [else (coinciden (cdr caso) (cdr list_e) valor)]
          )
```

El reconocimiento de patrones

Detectar patrón le pasan un valor, le pasan una lista de match y una lista de expresiones de que, si ese match es verdadero, pues se ejecuta la expresión y un ambiente para valorar las expresiones. Entonces, lo que hace es si comprobar los valores, si es una lista, pues se ejecuta lo de lista. Si es un null, ejecuta lo de null. Eso es lo que va a hacer él básicamente entonces. Él comprueba los valores de cada uno. Si es un array es porque es un vector entonces lo ejecuta, si es una lista pues porque es una lista, si está vacío, con null o comprueba si está vacío, si es un booleano o si es una expresión por defecto, lo ejecutaría tal cual. El cambio es que cuando es un número directamente él lo comprueba, pero si es un string él verifica que sea de tipo binario, hexadecimal, u octal y si es verdad, entonces lo que hace es tomar eso como un número, porque detecta que es un binario, aunque sea string, pues es un binario y lo mismo hace con las cadenas. Él verifica que sea una string, pero que no sea ni hexadecimal, ni binario, ni octal. Quita esos casos y evalúa la expresión de cada uno. Si es verdad que es de ese tipo que se pide, ya lo de asignación de variables a cada uno le extiende un ambiente donde se va a guardar el valor de esa variable que él pidió con el valor que vino y luego lo que hace es transformar vector que él recibe una lista de IDs. Entonces, lo que va a hacer es asignarle a cada uno de estos los valores y si ve que la lista es muy pequeña y hay muchos más valores, él asigna el resto al último y lo vuelve vector.

```
;; Reconocimiento de patrones
(expresion ("match" expresion "{" (arbno regular-exp "=>" expresion) "}") match-exp)
```

```
;match
(define detector_patron
  (lambda (valor primt_match expresi_match env)
    (cases regular-exp (car primt_match)
      (empty-match-exp ()
        (if (null? valor)
          (eval-expresion (car expresi_match) env)
          (detector_patron valor (cdr primt_match) (cdr expresi_match) env)
        )
```

```
(list-match-exp (cabeza cola)
  (if (list? valor)
    (eval-expresion (car expresi_match)
    (extend-env (cons cabeza (cons cola '())) (list (car valor) (cdr valor)) env))
    (detector_patron valor (cdr primt_match) (cdr expresi_match) env)
  )
```

```
(num-match-exp (ids)
  (if (number? valor)
    (eval-expresion (car expresi_match) (extend-env (list ids) (list valor) env))
    (if (string? valor)
      (cond
        [(or (equal? (string-ref valor 0) #\b)
         (and (equal? (string-ref valor 1) #\b) (equal? (string-ref valor 0) #\-)))
          (eval-expresion (car expresi_match) (extend-env (list ids) (list valor) env))
        ]
        [(or (equal? (string-ref valor 0) #\h)
         (and (equal? (string-ref valor 1) #\h) (equal? (string-ref valor 0) #\-)))
          (eval-expresion (car expresi_match) (extend-env (list ids) (list valor) env))
        ]
        [(or (equal? (string-ref valor 1) #\x)
         (and (equal? (string-ref valor 2) #\x) (equal? (string-ref valor 0) #\-)))
          (eval-expresion (car expresi_match) (extend-env (list ids) (list valor) env))
        ]
        [else (detector_patron valor (cdr primt_match) (cdr expresi_match) env)]
      )
      (detector_patron valor (cdr primt_match) (cdr expresi_match) env)
```

```
(cad-match-exp (ids)
  (if (string? valor)
    (cond
      [(not (or (equal? (string-ref valor 0) #\b)
                (and (equal? (string-ref valor 1) #\b)
                     (equal? (string-ref valor 0) #\-))))
      (not (or (equal? (string-ref valor 0) #\h)
                (and (equal? (string-ref valor 1) #\h)
                     (equal? (string-ref valor 0) #\-))))
      (not (or (equal? (string-ref valor 1) #\x)
                (and (equal? (string-ref valor 2) #\x)
                     (equal? (string-ref valor 0) #\-))))
      (eval-expresion (car expresi_match) (extend-env (list ids) (list valor) env))
      ]
      [else (detector_patron valor (cdr primt_match) (cdr expresi_match) env)]
    )
    (detector_patron valor (cdr primt_match) (cdr expresi_match) env)
  )
```

```
(bool-match-exp (ids)
  (if (boolean? valor)
    (eval-expresion (car expresi_match) (extend-env (list ids) (list valor) env))
    (detector_patron valor (cdr primt_match) (cdr expresi_match) env)
  )
)
(array-match-exp (ids)
  (if (vector? valor)
    (eval-expresion (car expresi_match) (extend-env ids (transfomar_vector ids valor 0) env))
    (detector_patron valor (cdr primt_match) (cdr expresi_match) env)
  )
)
(default-match-exp ()
  (eval-expresion (car expresi_match) env)
)
```

Como se representaron las estructuras

Se crea una función para buscar las estructuras, se guardan todas las estructuras en una variable que es el ambiente estructuras. Entonces, lo que va a hacer él es buscar el nombre de la estructura cuando lo evalúan en Eval-expresión, él busca el nombre de la estructura ahí y entonces es lo que va a hacer con la estructura. Es decir, bueno, la estructura es una lista de atributos junto con un vector de valores. Entonces, si él quiere modificar un valor, pues hay una función índice que lo ayuda a modificar ese valor y si, digamos, que él necesita consultar un atributo, él mira en la lista de atributos que la misma estructura tiene donde está ese atributo.

```
; funciones de estructuras

(define struct-decl->name
  (lambda (struct)
    (cases struct-decl struct
      (struct-exp (struct-name field-list)
        struct-name)
      (else (eopl:error "Invalid struct"))
    )
  )
)

(define struct-decl->attributes
  (lambda (struct)
    (cases struct-decl struct
      (struct-exp (struct-name field-list)
        field-list)
      (else (eopl:error "Invalid struct"))
    )
  )
```

```
; Actualiza el entorno de estructuras con nuevas declaraciones de estructuras
(define update-struct-env!
  (lambda (s-decls)
    (set! struct-env s-decls)))

(define lookup-struct
  (lambda (name)
    (let loop ((env struct-env))
      (cond
        [(null? env) (eopl:error "Unknown struct: " name)]
        [(eqv? (struct-decl->name (car env)) name) (car env)]
        [else (loop (cdr env))]
      )
```

```
; Encuentra el indice de un atributo en la lista de atributos de una estructura
(define find-attribute
  (lambda (attributes attr acc)
    (cond
      [(null? attributes) (eopl:error "Attribute not found: " attr)]
      [(equal? (car attributes) attr) acc]
      [else (find-attribute (cdr attributes) attr (+ acc 1))]
    )
  )
```