Compiladores 2020-1
Facultad de Ciencias UNAM
Práctica 6: Tablas de Símbolos

Jaime Arturo García Campos Alejandro Tonatiuh Valderrama Silva

10 de Noviembre, de 2019

Gramática fuente

La gramática que describe el lenguaje utilizado para el desarrollo de esta práctica es la siguiente:

```
<programa> ::= <expr>

<expr> ::= <const>
         | <var>
         | (begin <expr> <expr>*)
         | (const <type> <const>)
         | (primapp <prim> <expr>*)
         | (if <expr> <expr> <expr>)
         | (lambda ([<var> <type>]) <expr>)
         | (let ([<var> <type> <expr>])  <expr>)
         | (letrec ([<var> <type> <expr>]) <expr>)
         | (letfun ([<var> <type> <expr>]) <expr>)
         | (list <expr>*)
         | (<expr> <expr>)

<const> ::=  <boolean>
         | <integer>
         | <char>

<boolean> ::= #t | #f

<integer> ::= <digit> | <digit><integer>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<var> ::= <car> | <car><var> | <car><digit> | <car><digit><var>
```

```
<car> ::= a | b | c | ... | z

<prim> ::= + | - | * | / | length | car | cdr

<type> ::= Bool | Int | Char | List | Lambda | (<type> -> <type>)|(List
    of <type>)
```

Ejercicio 1

Implementamos el proceso **uncurry**, el cual recibe una expresión en L10 y devuelve una expresión en L11. Se encarga de descurrificar las expresiones lambda de nuestro lenguaje.

```
(define-pass uncurry : L10 (ir) -> L11 ()
  (definitions
   (define (uncurry var type body)
     (let* ([new-variables `([,var ,type])]
            [new-body (unparse-L10 body)]
            [lambda-parts (uncurry-aux new-variables new-body)])
       `(lambda ,(first lambda-parts) ,(second lambda-parts))))
   (define (uncurry-aux new-variables new-body)
      (cond
        [(equal? 'lambda (car new-body))
         (uncurry-aux `,(append new-variables (cadr new-body)) `,(caddr
             new-body))]
        [else (append (list new-variables) (list new-body))])))
  (Expr : Expr (ir) -> Expr ()
        [(lambda ([,x ,t]) ,body) (parser-L11 (uncurry x t body))]
        [else ir]))
```

Ejercicio 2

Definimos la función **symbol-table-var**que genera la tabla de símbolos de una expresión del lenguaje. Este proceso recibe una expresión en L11 y devuelve una un hash que representa la tabla de símbolos, donde tiene como llave el identificador de la variable y como valor un par que almacena el tipo de la variable y su valor.

```
(define (symbol-table-var exp)
  (nanopass-case (L11 Expr) exp
         [(begin ,e* ... ,e) (begin
                               (symbol-table-var e)
                               (map symbol-table-var e*)
                               symbol-table)]
         [(primapp ,pr ,e* ... ,e0) (begin
```

```
                                (symbol-table-var e0)
                               (map symbol-table-var e*)
                               symbol-table)]
        [(if ,e0 ,e1 ,e2) (begin
                            `,(if e0 (symbol-table-var e1) (
                               symbol-table-var e2))
                            symbol-table)]
        [(lambda ([,x ,t]) ,body) (begin
                                    (symbol-table-var body)
                                    symbol-table)]
        [(let ([,x ,t ,e]) ,[body])    (begin
                                         `,(hash-set! symbol-table x (cons t
                                            e))
                                         symbol-table)]
        [(letrec ([,x ,t ,e]) ,[body]) (begin
                                         (symbol-table-var e)
                                         `,(hash-set! symbol-table x (cons t
                                            e))
                                         symbol-table)]
        [(letfun ([,x ,t ,e]) ,[body]) (begin
                                         (symbol-table-var e)
                                         `,(hash-set! symbol-table x (cons t
                                            e))
                                         symbol-table)]
        [(list ,e* ...) (begin
                          (map symbol-table-var e*)
                          symbol-table)]
        [(,e0 ,e1) (begin
                     (symbol-table-var e0)
                     (symbol-table-var e1)
                     symbol-table)]
        [else symbol-table]))
```

Ejercicio 3

Implementamos el proceso **assignment**, que modifica los constructores **let, letrec** y **letfun**, eliminando el valor asociado a los identificadores y el tipo correspondiente. Es decir, sólo regresaremos el constructor y su **body**.
Este proceso recibe una expresión de L11 y devuelve una de L12 y la tabla de símbolos correspondiente a la expresión. Lo que implica que a partir de este proceso siempre cargaremos con la tabla de símbolos entre los procesos.

```
(define-pass assignment : L11 (ir) -> L12 (table)
  (Expr : Expr (ir) -> Expr ()
        [(let ([,x ,t ,e]) ,[body])    `(let ,body)]
```

```
          [(letrec ([,x ,t ,[e]]) ,[body]) `(letrec ,body)]
          [(letfun ([,x ,t ,[e]]) ,[body]) `(letfun ,body)])
  (values (Expr ir) (symbol-table-var ir)))
```

Lenguajes Usados

Definición de L10

Este es el lenguaje que se toma como punto de partida para esta práctica, que es la salida de la práctica anterior. En ***nano-pass***[1]:

```
(define-language L10
  (terminals
   (variable (x))
   (primitive (pr))
   (constant (c))
   (type (t)))
  (Expr (e body)
    x
    (const t c)
    (begin e* ... e)
    (primapp pr e* ... e0)
    (if e0 e1 e2)
    (lambda ([x t]) body)
    (let ([x t e]) body)
    (letrec ([x t e]) body)
    (letfun ([x t e]) body)
    (list e* ...)
    (e0 e1)))
```

Definición de L11

En ***nano-pass***:

```
(define-language L11
  (extends L10)
  (Expr (e body)
    (- (lambda ([x t]) body))
    (+ (lambda ([x* t*] ...) body))))
```

[1]La definición formal está al inicio del documento

Definición formal:

```
<programa> ::= <expr>

<expr> ::= <const>
         | <var>
         | (begin <expr> <expr>*)
         | (const <type> <const>)
         | (primapp <prim> <expr>*)
         | (if <expr> <expr> <expr>)
         | (lambda ([<var>* <type>*]) <expr>)
         | (let ([<var> <type> <expr>])  <expr>)
         | (letrec ([<var> <type> <expr>]) <expr>)
         | (letfun ([<var> <type> <expr>]) <expr>)
         | (list <expr>*)
         | (<expr> <expr>)

<const> ::=  <boolean>
         | <integer>
         | <char>

<boolean> ::= #t | #f

<integer> ::= <digit> | <digit><integer>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<var> ::= <car> | <car><var> | <car><digit> | <car><digit><var>

<car> ::= a | b | c | ... | z

<prim> ::= + | - | * | / | length | car | cdr

<type> ::= Bool | Int | Char | List | Lambda | (<type> -> <type>)|(List
    of <type>)
```

Definición de L12

En ***nano-pass***:

```
(define-language L12
  (extends L11)
  (Expr (e body)
    (- (let ([x t e]) body)
```

```
        (letrec ([x t e]) body)
        (letfun ([x t e]) body))
    (+ (let body)
        (letrec body)
        (letfun body))))
```

Definición formal:

```
<programa> ::= <expr>

<expr> ::= <const>
         | <var>
         | (begin <expr> <expr>*)
         | (const <type> <const>)
         | (primapp <prim> <expr>*)
         | (if <expr> <expr> <expr>)
         | (lambda ([<var>* <type>*]) <expr>)
         | (let  <expr>)
         | (letrec <expr>)
         | (letfun <expr>)
         | (list <expr>*)
         | (<expr> <expr>)

<const> ::=  <boolean>
         | <integer>
         | <char>

<boolean> ::= #t | #f

<integer> ::= <digit> | <digit><integer>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<var> ::= <car> | <car><var> | <car><digit> | <car><digit><var>

<car> ::= a | b | c | ... | z

<prim> ::= + | - | * | / | length | car | cdr

<type> ::= Bool | Int | Char | List | Lambda | (<type> -> <type>)|(List
    of <type>)
```

Predicados

```
;;PREDICADOS

(define symbol-table (make-hash))

(define fun-count 0)

(define (variable? x) (and (symbol? x) (not (primitive? x)) (not (
   constant? x))))

(define (primitive? x) (memq x '(+ - * / length car cdr)))

(define (constant? x)
  (or (integer? x)
      (char? x)
      (boolean? x)))

;; SISTEMA DE TIPOS
;; Int | Char | Bool | Lambda | List | (List of T) | (T     T)
(define (type? x) (or (b-type? x) (c-type? x)))
(define (b-type? x) (memq x '(Bool Char Int List String Lambda)))
(define (c-type? x) (if (list? x)
   (let* (
      [f (car x)]
      [s (cadr x)]
      [t (caddr x)])
   (or (and (equal? f 'List) (equal? s 'of) (type? t))
      (and (type? f) (equal? s '   ) (type? t))))
   #f))

(define (arit? x) (memq x '(+ - * /)))

(define (lst? x) (memq x '(length car cdr)))
```

Comentarios

- No entendemos porque para hacer la recursión sobre las expresiones de los constructores que pueden agregar nuevas variables no se funciona usar los catamorfismos. Es decir, no podemos hacer:

```
[(letrec ([,x ,t ,[e]]) ,[body]) (begin
                `,(hash-set! symbol-table x (cons t e))
                symbol-table)]
```

Si no que tenemos que hacer la recursión explicitamente, es decir:

```
[(letrec ([,x ,t ,e]) ,[body]) (begin
                (symbol-table-var e)
                `,(hash-set! symbol-table x (cons t e))
                symbol-table)]
```

- Nos quedó la duda de como manejar el **body** en el ejercicio 2 para poder "desmenuzarlo" tratarlo como una expresión del lenguaje para para hacer el match como lo hemos venido haciendo. Nosotros en cambio tratamos la expresión del **body** como si fuera una lista.

.