Compiladores 2020-1
Facultad de Ciencias UNAM
Práctica 4: Análisis Sintácticp

Arturo García Campos Alejandro Valderrama Silva

06 de octubre de 2019

Desarrollo

Antes de iniciar el desarrollo de esta práctica, recuperamos las funciones y predicados declarados en la práctica 3. Esto para continuar con la implementación de un parser que poco a poco va transformando tipos en expresiones.

Gramática Original

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

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

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

<list> ::= empty | (cons <const> <list>)

<string> ::= "" | " <char> <string> "

<char> ::=  a | b | c | ... | z | ... | @ | # | $ | % | & | ...

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

<type> ::= Bool | Int | Char | List | String
```

Ejercicio 1

Para este ejercicio lo que hicimos fue agregar una función auxiliar que nos construyera una expresión del tipo let/letrec correspondiente a los valores ingresados del operador del mismo tipo. Así sólo nos tenemos que preocupar por construir la recursión de manera correcta en el catamorfismo.

```
(define-pass curry-let : L6(ir) -> L7 ()
  (definitions
    (define (curry vars types exps body op)
      (cond
        [(empty? vars) (unparse-L7 body)]
        [else `(,op ([,(car vars) ,(car types) ,(unparse-L7 (car exps))
           ])
               ,(curry (cdr vars) (cdr types) (cdr exps) body op))]
        )))
  (Expr : Expr (ir) -> Expr ()
        [(let ([,x* ,t* ,[e*]] ... ) ,[body])
         (parser-L7 (curry x* t* e* body 'let))]
        [(letrec ([,x* ,t* ,[e*]] ... ) ,[body])
         (parser-L7 (curry x* t* e* body 'letrec))]))
```

Ejercicio 2

Con este ejercicio queremos cambiar todos los *let* utilizados para definir funciones por *letrec*. Lo único que identificar cuando pasa eso es detectar cuando se está definiendo una *lambda*.

```
(define-pass identify-assigments : L7(ir) -> L7()
  (Expr : Expr (ir) -> Expr ()
        [(let ([,x ,t ,[e]]) ,[body])
         (if (equal? t 'lambda)
             `('letrec ([,x ,t ,e]) ,body)
             ir)]))
```

Ejercicio 3

Para este ejercicio lo que hicimos fue creador un contador global para poder asegurar que cada nombre nuevo para funciones anónimas será único en el programa. Luego de eso basta crear una nueva expresión, tipo let ($letfun$), que nos describa esa función anónima, donde el cuerpo será exactamente el mismo que hemos recibido; de esta manera no cambia en nada.

```
(define fun-count 0)
```

```
(define-pass un-anonymous : L7(ir) -> L8 ()
  (definitions
    (define (new-name)
      (begin
        (define str (string-append "foo" (number->string fun-count)))
        (set! fun-count (+ 1 fun-count))
        (string->symbol str))))
  (Expr : Expr (ir) -> Expr ()
        [(lambda ([,x ,t]) ,[body])
         (let ([name (new-name)])
           `(letfun ([,name Lambda ,(parser-L8 (unparse-L7 ir))]) ,name
             ))]))
```

Ejercicio 4

Para este ejercicio nosotros no hicimos un proceso verificador aparte. Por como construimos el preproceso "$eta - expand$" fue directo establecer una forma, en dicho proceso, de manejar los errores de aridad dentro de éste.
Por lo este ejercicio queda .ªbsorbido"dentro del proceso "$eta - expand$" de la siguiente manera:

```
(define-pass eta-expand : L4 (ir) -> L5 ()
  (definitions
    (define binarios #hash(
                       (+ . ([x0 Int] [x1 Int]))
                       (- . ([x0 Int] [x1 Int]))
                       (* . ([x0 Int] [x1 Int]))
                       (/ . ([x0 Int] [x1 Int])))))
    (define unarios #hash(
                       (length . ([x0 List]))
                       (car . ([x0 List]))
                       (cdr . ([x0 List])))))
  (Expr : Expr (ir) -> Expr ()
        [(,pr) `,(if (hash-has-key? binarios `,pr)
                     `((lambda ,(hash-ref binarios `,pr) (primapp ,pr x0 x1)))
                     `((lambda ,(hash-ref unarios `,pr) (primapp ,pr x0))))]
        [((,pr) ,[e0]) `,(if (hash-has-key? unarios `,pr)
                             `((lambda ,(hash-ref unarios `,pr) (primapp ,pr x0)) ,e0)
                             (error "Arity in" `,pr  "must be 2"))]
        [((,pr) ,[e0] ,[e1]) `,(if (hash-has-key? binarios `,pr)
                                   `((lambda ,(hash-ref binarios `,pr) (primapp ,pr x0 x1)) ,e0, e1)
                                   (error "Arity in" `,pr "must be 1"))]))
```

Ejercicio 5

Para este ejercicio definimos una función auxiliar que nos de una lista con todas las variables libres de la expresión, es lo hacemos con una función auxiliar de nanopass llamada *nanopass − case* que básicamente nos deja hacer un tipo *cond* entre una expresión y expresiones de la gramática.

```
    (define (FV exp)
  (nanopass-case (L8 Expr) exp
                 [,x `(,x)]
                 [(cuote ,q) '()]
                 [(begin ,e* ... ,e) (append (FV e) (foldr append '() (
                    map FV e*)))]
                 [(primapp ,pr ,e* ... ,e0) (foldr append '() (map FV e
                    *))]
                 [(list ,e* ...) (foldr append '() (map FV e*))]
                 [(if ,e0 ,e1 ,e2) (append (FV e0) (FV e1) (FV e2))]
                 [(lambda ([,x* ,t*]) ,body) (remove (FV body) x*)]
                 [(let ([,x ,t ,e]) ,body) (remove (append (FV body) (
                    FV e)) x)]
                 [(letrec ([,x ,t ,e]) ,body) (remove (append (FV body)
                     (FV e)) x)]
                 [(letfun ([,x ,t ,e]) ,body) (remove (append (FV body)
                     (FV e)) x)]
                 [else '("Something went wrong")]
                 ))
```

```
(define-pass verify-vars : L8 (ir) -> L8 ()
```

```
  (Expr : Expr (ir) -> Expr ()
        [else (if (empty? (FV ir))
                  ir
                  (error "There are free variables in your expression")
                    )]))
```

Lenguajes Usados

L7

```
(define-language L7
  (extends L6)
  (Expr (e body)
   (- (let ([x* t* e*] ...) body)
      (letrec ([x* t* e*] ...) body))
   (+ (let ([x t e]) body)
      (letrec ([x t e]) body)
      (let () body)
      (letrec () body))))
```

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

<expr> ::= <list>
         | <string>
         | (cuote <quotable>)
         | (<primitiva> <const> <const>*)
         | (begin <expr> <expr>*)
         | (if <expr> <expr>)
         | (if <expr> <expr> <expr>)
         | (lambda ([<var> <type>]) <expr>)
         | (let ([<var> <type> <expr>])  <expr>)
         | (letrec ([<var> <type> <expr>]) <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

<list> ::= empty | (cons <const> <list>)

<string> ::= "" | " <char> <string> "

<char> ::=  a | b | c | ... | z | ... | @ | # | $ | % | & | ...

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

<type> ::= Bool | Int | Char | List | String

<quotable> ::= q
```

L8

```
(define-language L8
  (extends L7)
  (Expr (e body)
        (+ (letfun ([x t e]) body))))
```

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

<expr> ::= <list>
         | <string>
         | (cuote <quotable>)
         | (<primitiva> <const> <const>*)
         | (begin <expr> <expr>*)
         | (if <expr> <expr>)
         | (if <expr> <expr> <expr>)
         | (lambda ([<var> <type>]) <expr>)
         | (let ([<var> <type> <expr>])  <expr>)
         | (letrec ([<var> <type> <expr>]) <expr>)
         | (letfun   ([<var> <type> <expr>]) <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

<list> ::= empty | (cons <const> <list>)

<string> ::= "" | " <char> <string> "

<char> ::=  a | b | c | ... | z | ... | @ | # | $ | % | & | ...

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

<type> ::= Bool | Int | Char | List | String

<quotable> ::= q
```

Predicados

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

(define (primitive? x) (memq x '(+ - * / length car cdr and or not)))

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

(define (type? x) (memq x '(Bool Char Int List String Lambda)))

(define (primitiva? pr) (memq pr '(+ - * / length car cdr)))

(define (quotable? q) (or (constant? q) (and (symbol? q) (not (
   primitive? q)))))
```

Comentarios

Donde tuvimos más problemas y nos trabamos más, fue en el ejercicio 3 porque no entendemos porque tenemos que hacer lo siguiente:

```
‘(letfun ([,name Lambda ,(parser-L8 (unparse-L7 ir))]) ,name))
```

En vez de simplemente:

```
‘(letfun ([,name Lambda ,ir]) ,name))
```

Para regresar tal cual la expresión que recibimos en dicho preproceso.