Universidad Nacional Autónoma de México
Facultad de Ciencias
Lenguajes de Programación

Práctica 3

Karla Ramírez Pulido
karla@ciencias.unam.mx

J. Ricardo Rodríguez Abreu
ricardo_rodab@ciencias.unam.mx

Manuel Soto Romero
manu@ciencias.unam.mx

Luis A. Patlani Aguilar
ayudantefc@gmail.com

Fecha de inicio: 6 de septiembre de 2017
Fecha de término: 20 de septiembre de 2017
Semestre 2018-1

Objetivos

- Comprender los tipos de análisis por los que pasa un código fuente antes de generar un código ejecutable.
- Implementar un analizador sintáctico y un analizador semántico para el lenguaje `WAE`.

Descripción general

La práctica consiste en implementar un pequeño intérprete que permita mostrar los tipos de análisis requeridos para generar código ejecutable dado el código fuente. La gramática en EBNF para las expresiones del lenguaje `WAE` (*With Arithmetic Expressions*), que se implementará en esta práctica es la siguiente:

```
<expr> ::= <id>
         | <num>
         | {<op> <expr>+}
         | {with {{<id> <expr>}+} <expr>}
         | {with* {{<id> <expr>}+} <expr>}

<id> ::= a |... | z | A | ... | Z | aa | ab | ... | aaa | ...
         (Cualquier combinación de caracteres alfanuméricos
          con al menos uno alfabético)

<num> ::= ... | -2 | -1 | 0 | 1 | 2 | ...
         (Cualquier número admitido por Racket)

<op> :: + | - | * | / | % | min | max | pow
```

Archivos requeridos

El material de esta práctica consta de los siguientes archivos[1]:

- `grammars.rkt` archivo con la definición de los tipos de datos abstractos que definen los ASA para el lenguaje `WAE`.
- `parser.rkt` archivo dónde se implementará el analizador sintáctico para el lenguaje `WAE`.
- `interp.rkt` archivo dónde se implementará el analizador semántico para el lenguaje `WAE`.
- `practica3.rkt` archivo con la lógica necesaria para ejecutar el intérprete final de `WAE`.
- `test-practica3.rkt` archivo dónde se agregarán las pruebas unitarias de la práctica

Desarrollo de la práctica

Ejercicio 3.1 (1.5 pts.) Escribir en el archivo `readme.txt` expresiones aceptadas por la gramática del lenguaje `WAE` de acuerdo a lo siguiente:

1. Al menos dos identificadores válidos.
2. Al menos cinco números válidos:
   a) Un entero.
   b) Un entero negativo.
   c) Un racional.
   d) Un flotante.
   e) Un complejo.
3. Al menos diez operaciones válidas:
   a) Una expresión por cada operación posible que use al menos tres operandos.
   b) Combinaciones de todas las operaciones posibles.
4. Al menos tres expresiones `with` válidas:
   a) Una expresión sencilla con identificador, valor y como cuerpo una expresión aritmética.
   b) Una expresión con identificador, valor y como cuerpo otra expresión `with`.
   c) Una expresión con identificador, como valor una expresión `with` y como cuerpo la anidación tres `with`.
5. Al menos tres expresiones `with*` válidas: se debe cumplir lo mismo que en el punto 3 que además permita definir identificadores en términos de otros previamente definidos, por ejemplo: `{with* {{a 2} {b a}} {+ b b}}`.

[1]Los archivos pueden descargarse desde la página del curso `http://lenguajesfc.com`.

Ejercicio 3.2 (0.5 pts.) El analizador léxico recibe una expresión y las separa en lexemas. En Racket podemos ahorrarnos este análisis al usar una de sus primitivas ¿cuál es esta primitiva?, ¿cómo funciona?, ¿por qué es útil para realizar el análisis léxico? Contestar estas preguntas en el archivo `readme.txt`.

Ejercicio 3.3 (3 pts.) En el análisis sintáctico, a partir de una lista de lexemas, se construye el Árbol de Sintaxis Abstracta (ASA) correspondiente. En el archivo `grammars.rkt` se encuentra el TDA que define los constructores para realizar este mapeo:

```
;; TDA para representar el árbol de sintaxis abstracto del
;; lenguaje WAE.
(define-type WAE
   [id (i symbol?)]
   [num (n number?)]
   [op (f procedure?) (args (listof WAE?))]
   [with (bindings (listof binding?)) (body WAE?)]
   [with* (bindings (listof binding?)) (body WAE?)])

;; TDA para asociar identificadores con valores.
(define-type Binding
   [binding (name symbol?) (value WAE?)])
```

1. Por cada una de las expresiones definidas en el Ejercicio 3.1, agregar una prueba unitaria al archivo `test-practica3.rkt` que indique cómo se debe transformar dicha expresión después de realizar el análisis sintáctico (mediante la función `parse`). Por ejemplo, la expresión `'{+ 1 2}` se transforma a `(op + (list (num 1) (num 2)))` con lo cual se escribiría la prueba:

   ```
   (test (parse '{+ 1 2}) (op + (list (num 1) (num 2))))
   ```

2. Completar el cuerpo de la función `(parse sexp)` del archivo `parser.rkt` para que realice el análisis sintáctico correspondiente, es decir, construye expresiones del TDA `WAE`.

   Todas las operaciones se deben mapear a funciones de Racket; sin embargo, el operador `pow` no puede mapearse directamente a `expt` pues este último no es multiparámetrico, de esta forma, se debe definir una función `mexpt` en el archivo `parser.rkt` que implemente la potencia multiparamétrica.

   ```
   ;; parse: s-expresion -> WAE.
   (define (parse sexp)
      ...)
   ```

   ```
   > (parse '{+ 1 2})
   (op + (list (num 1) (num 2)))
   ```

Ejercicio 3.4 (5 pts.) El analizador semántico recibe un árbol de sintaxis abstracta (ASA) y regresa su evaluación.

1. Por cada una de las expresiones definidas en el Ejercicio 3.1, agregar una prueba unitaria al archivo `test-practica3.rkt` que indique cómo se debe evaluar dicha expresión después de realizar tanto el análisis sintáctico como el semántico (mediante las funciones `parse` e `interp` respectivamente). Por ejemplo, la expresión `'{+ 1 2}` debe transformarse a 3 con lo cual se escribiría la prueba:

   ```
   (test (interp (parse '{+ 1 2})) 3)
   ```

2. Para evaluar expresiones `with` es necesario aplicar sustituciones textuales. Completar el cuerpo de la función `(subst expr sub-id val)` del archivo `interp.rkt` para que implemente el algoritmo de sustitución textual sobre expresiones de `WAE`.

   ```
   ;; subst: WAE symbol WAE -> WAE
   (define (subst expr sub-id val)
     ...)
   ```

   ```
   > (subst (op + (list (id 'a) (num 3))) 'a (num 4))
   (op + (list (num 4) (num 3)))
   ```

3. Completar el cuerpo de la función `(interp exp)` del archivo `interp.rkt` para que realice el análisis semántico correspondiente, es decir, evaluar expresiones de `WAE`. Tomar los siguientes puntos a consideración:

   - Los identificadores por sí mismos no pueden ser evaluados, por lo que se debe regresar un error describiendo que se tiene un identificador libre. Ejemplo:

     ```
     > (interp (parse 'foo))
     error: Identificador libre
     ```

   - Los números se evalúan a sí mismos. Ejemplo:

     ```
     > (interp (parse '1729))
     1729
     ```

   - Los operadores son n-arios, por lo que esta versión del intérprete tiene un constructor op que recibe una función con la cual realiza la operación definida a cada uno de sus parámetros. Ejemplo:

     ```
     > (interp (parse '{+ 1 2 3 4 5}))
     15
     ```

   - Las expresiones `with` son multiparamétricas, lo cual quiere decir que tienen más de un identificador. Ejemplo:

     ```
     > (interp (parse '{with {{a 2} {b 3}} {+ a b}}))
     5
     ```

- Las expresiones `with*` presentan un comportamiento parecido al de la primitiva `with`, sin embargo, estas expresiones permiten definir identificadores en términos de otros definidos anteriormente. Por ejemplo: `{with* {{a 2} {b {+ a a}}} b}`. Se interpreta similar a `with`, la única diferencia es que también se deben procesar los identificadores. Ejemplo:

```
> (interp (parse '{with* {{a 2} {b {+ a a}}} b}))
4
```

```
;; interp: WAE -> number
(define (interp exp)
   ...)
```

```
> (interp (op + (list (num 1) (num 2))))
3
```

Ejercicio 3.5 (0 pts.) Una vez finalizados los ejercicios anteriores, ejecutar el archivo `practica3.rkt`. Realizar pruebas significativas para verificar que la práctica se completó con éxito.

```
Bienvenido a DrRacket, versión 6.8 [3m].
Lenguaje: plai, with debugging; memory limit: 128 MB.
(λ) {+ 1 2 3}
6
(λ) {with {{a 2} {b 3}} {+ a b}}
5
(λ) {with* {{a 2} {b {+ a a}}} b}
4
(λ) {exit}
>
```

Figura 1: *Ejecución de* `WAE`

Puntaje total: *10 puntos*

Entrega

Los archivos que se deben enviar a los ayudantes de laboratorio son:

- `parser.rkt`
- `interp.rkt`
- `test-practica3.rkt`

Recordando seguir los lineamientos de entrega de prácticas especificados en la sección correspondiente de la página del curso: `http://lenguajesfc.com/lineamientos.html`.

Puntos extra

Escoger alguno de los siguientes ejercicios y resolverlo **individualmente**[2]:

Punto extra 3.1 (1 pt.) En esta práctica se mencionaron tres de los tipos análisis por los que pasa el código fuente, sin embargo, en las clases de teoría se mencionó que existe una etapa de optimizaciones. Investigar algún tipo de optimización que realicen lo compiladores o intérpretes modernos y escribir un ensayo de al menos dos cuartillas. El ensayo debe incluir: Título, Introducción, Desarrollo, Conclusiones y Bibliografía. No se tomará en cuenta ningún ensayo que no esté debidamente citado e incluya las referencias correspondientes. El ensayo deberá entregarse de forma impresa a más tardar el 20 de septiembre de 2017 durante la sesión de laboratorio.

Punto extra 3.2 (1 pt.) En el ejercicio 3.2 se menciona que no es necesario escribir un analizador léxico pues Racket provee una primitiva especial. El punto extra consiste en escribir una función `(lexer s)` que tome una cadena y la procese igual que esta primitiva. Por ejemplo `(lexer ‘‘{+ 1 2 3}’’)` debe regresar `'{+ 1 2 3}`. El ejercicio debe enviarse a más tardar el 20 de septiembre de 2017 en un archivo `cuenta_p3.rkt` donde `cuenta` es el número de cuenta del alumno, al correo `manu+ldp@ciencias.unam.mx` con el asunto `[LDP-Extra P3]`.

Referencias

Algunas referencias de consulta:

[1] Karla Ramírez, Manuel Soto, Ricardo Rodríguez, *Notas de laboratorio del curso de Lenguajes de Programación*, Semestre 2018-1, Facultad de Ciencias, UNAM. Disponibles en: [`http://lenguajesfc.com/notas.html`].

[2] Rodrigo Ruiz Murgía, *Manual de prácticas para la asignatura de Lenguajes de Programación*, Reporte de actividad docente, Facultad de Ciencias, 2016.

[3] Shriram Krishnamurthi, *Programming Languages: Application and Interpretation*, Primera edición, Brown University, 2007.

[2] Pueden enviarse ambos ejercicios para ser revisados, sin embargo, sólo uno será tomado en cuenta.