Lenguajes de Programación, 2017-2

Práctica 3: Generación de código ejecutable

Manuel Soto Romero

Fecha de inicio: 3 de marzo de 2017
Fecha de término: 17 de marzo de 2017

Instrucciones

- Completar de manera clara y ordenada las funciones de los archivos `parser.rkt` e `interp.rkt`.
- La práctica debe ejecutarse sin errores ni advertencias. Usar el archivo `practica3.rkt` para comprobar la correcta ejecución de la práctica.
- La entrega es por equipos de máximo tres integrantes. Seguir los lineamientos especificados en: `http://lenguajesfc.com/lineamientos.html`.
- Los puntos extra son individuales y sólo se puede escoger uno de éstos. El punto extra teórico debe de realizarse **a computadora y entregarse impreso en hojas blancas o recicladas a más tardar el 17 de marzo durante la sesión de laboratorio**. El punto extra de programación se envía por correo electrónico en un archivo `<no.cuenta>_P03.rkt` con el asunto `[LDP-EXP3]`. Incluir el nombre completo del autor en el cuerpo del correo.

Descripción general

La práctica consiste en implementar un pequeño intérprete que permita mostrar los tipos de análisis que se requieren para generar código ejecutable dado el código fuente. La gramática en EBNF para las expresiones del lenguaje `WAE` (*With Arithmetic Expression*), que revisaremos 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 | ...

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

A manera de repaso, veamos cómo se interpreta la expresión `{with {{a 2} {b 3}} {+ a b}}`:

- **Análisis léxico**

  Lo primero, es separar en lexemas la expresión anterior. En Racket usamos `quote` para obtener los lexemas:

  `(quote {with {{a 2} {b 3}} {+ a b}}) = '(with ((a 2) (b 3)) (+ a b))`

  Esto nos permite procesar la expresión anterior como una lista de símbolos.

- **Análisis sintáctico**

  El siguiente paso consiste en convertir la expresión de sintaxis concreta a su representación como árbol de sintaxis abstracta:

  ```
  (parse '(with ((a 2) (b 3)) (+ a b))) =
  (with (list (binding 'a (num 2)) (binding 'b (num 3))) (op + (list (id 'a) (id 'b))))
  ```

- **Análisis semántico**

  Por último, le asociaremos un significado al árbol de sintaxis abstracta correspondiente para posteriormente evaluar la expresión que hayamos obtenido. En este caso, tenemos que sustituir los identificadores del `with` en su cuerpo y después evaluarlo:

  ```
  (subst (subst (op + (list (id 'a) (id 'b))) 'a (num 2)) 'b (num 3)) =
  (op + (list (num 2) (num 3)))
  ```

  para que `(interp (op + (list (num 2) (num 3))))` nos regrese el valor de `5`

Ejercicios

1. **Análisis sintáctico**

   Anexo a este PDF se encuentra un archivo `parser.rkt` que contiene la firma de una función `(parse sexp)` la cual toma una expresión en sintaxis concreta y regresa su representación en sintaxis abstracta. El tipo de dato abstracto `WAE` con la estructura del árbol de sintaxis abstracta se encuentra en el archivo `grammars.rkt`.

   Modificar esta función para que procese expresiones del usuario y las convierta en expresiones de `WAE`.

2. **Análisis semántico**

   Anexo a este PDF se encuentra un archivo `interp.rkt` que contiene la firma de una función `(interp exp)` que recibe un árbol de sintaxis abstracta y regresa la interpretación de ese árbol.

   Modificar esta función para que procese expresiones de `WAE`. Tomar los siguientes puntos a consideración:

   - 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[1]. Ejemplo:

[1] Para interpretar estas expresiones, se recomienda ampliamente hacer uso de la función `apply`. Para la operación `pow`, se recomienda escribir una función propia en lugar de `expt`.

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

- Las expresiones `with` son multiparamétricas, lo cual quiere decir que tienen más de un identificador[2]. Ejemplo:

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

- Las expresiones `with*` presentar un comportamiento de 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 implementa similar a `with`, la única diferencia es que también se deben procesar los identificadores[3]. Ejemplo:

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

Una vez que se hayan completado ambas funciones, se debe de ejecutar el archivo `practica3.rkt` para comprobar que los ejercicios se resolvieron correctamente.

Puntos extra

- (1 pt.) En esta práctica se mencionaron tres tipos de los análisis por los que pasa el código fuente, sin embargo, en las clases de teoría se mencionó que existe una etapa de optimizaciones. Investigar y escribir un ensayo de al menos dos cuartillas donde se hable de algún tipo de optimización que realicen los compiladores o intérpretes modernos.

- (1 pt.) A lo largo del curso se ha dicho que escribir un analizador léxico no es necesario pues Racket provee la primitiva `quote`. El punto extra consiste en escribir una función `(lexer s)` que tome una cadena y los procese igual que `quote`. Por ejemplo:

```
> (lexer ``1'')
'1
> (lexer ``{+ 1 2 3}'')
'{+ 1 2 3}
> (lexer ``{with {{a 2} {b 3}} {+ a b}}'')
'(with ((a 2) (b 3)) (+ a b))
```

[2]Para interpretar estas expresiones **se debe usar el algoritmo de sustitución**. Se recomienda altamente hacer uso `map` y `filter`.

[3]Se sugiere hacer uso de `foldr`