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

Fecha de inicio: 2 de marzo de 2018
Fecha de término: 16 de marzo de 2018
Semestre 2018-2

Objetivo

Reforzar (1) los conceptos relacionados con los tipos de análisis por los que pasa el código fuente para generar código ejecutable e (2) implementar un analizador sintático y semántico para el lenguaje de programación `WAE`.

Antecedentes

En las sesiones previas de laboratorio se revisaron los pasos para generar código ejecutable dado el código fuente y se realizaron actividades para implementar un analizador sintáctico y un analizador semántico para una versión simplificada del lenguaje de programación `WAE`. Se recomienda revisar dichas actividades junto con las Notas 2 del curso.

Repositorio

El material necesario para completar esta práctica se encuentra en el repositorio de *GitHub Classroom* del curso: `https://classroom.github.com/g/3NSsYZOq`.

Desarrollo de la práctica

La gramática en EBNF para las expresiones del lenguaje `WAE` (*With and Arithmetic Expressions)* que se implementará en esta práctica es la siguiente:

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

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

 <num> ::= ... | -1 | 0 | 1 | 2 | ...

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

En equipos de **tres integrantes** completar las funciones faltantes de los archivos `parser.rkt` e `interp.rkt` hasta que pasen todas las pruebas unitarias incluidas en el archivo `pruebas_practica3.rkt` y se ejecute correctamente el archivo `pratica3.rkt`[1].

Ejercicio 3.1 (2.5 pts.) Completar el cuerpo de la función `(parse sexp)` del archivo `parser.rkt` que realiza el análisis sintáctico correspondiente, es decir, construye expresiones del TDA `WAE` incluido en el archivo `grammars.rkt`.

Todas las operaciones se deben mapear a funciones de Racket mediante la función `elige` previamente definida en el archivo `parser.rkt` para que el mapeo funcione, se debe completar el cuerpo de las funciones `mmodulo` y `mexpt` del archivo `grammars.rkt` que implementan el módulo y la potencia multiparamétrica.

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

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

Ejercicio 3.2 (2.5 pts.) Para evaluar expresiones `with` es necesario aplicar sustituciones textuales. Completar el cuerpo de la función `(subst expr sub-id val)` del archivo `interp.rkt` que implementa el algoritmo de sustitución textual a 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 4)))
```

[1]Para tener derecho a calificación, los archivos deben ejecutarse sin errores.

Ejercicio 3.3 (5 pts.) Completar el cuerpo de la función `(interp expr)` del archivo `interp.rkt` que realiza 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 indicando 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 operandos. 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, restas 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 se también se deben procesar los identificadores. Ejemplo:

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

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

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

Referencias

Algunas referencias de consulta:

[1] Karla Ramírez, Manuel Soto, *Notas de laboratorio del curso de Lenguajes de Programación*, Semestre 2018-12, 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.