Tarea 7

Lenguajes de Programación

Eduardo Acuña Yeomans[a]

29 de octubre de 2023

Considera la implementación y especificación del lenguaje LET anexos en el archivo `let-lang.zip` para resolver los siguientes problemas. Los términos *extender*, *incorporar*, *agregar* y *modificar* el lenguaje se refieren a la especificación formal. Deberás modificar la implementación únicamente cuando se especifique explícitamente.

En tu tarea, anexa todos los archivos de la implementación de LET con las modificaciones que hayas realizado, junto con un archivo de LaTeX llamado `tarea-07.tex` y su versión compilada `tarea-07.pdf` donde describas tu trabajo de cada ejercicio en el orden que se muestra en esta tarea.

1. Extiende el lenguaje agregando un nuevo operador `minus` que toma un argumento n y regresa $-n$. Por ejemplo, el valor de `minus(-(minus(5),9))` debe ser 14.
2. Extiende el lenguaje agregando operadores para la suma, multiplicación y cociente de enteros.
3. Agrega un predicado de igualdad numérica `equal?` y predicados de orden `greater?` y `less?` al conjunto de operaciones del lenguaje LET.
4. Agrega operaciones de procesamiento de listas al lenguaje, incluyendo `cons`, `car`, `cdr`, `null?` y `emptylist`. Una lista debe poder contener cualquier valor expresado, incluyendo otra lista. Por ejemplo,

   ```
   let x = 4
   in cons(x,
           cons(cons(-(x,1),
                     emptylist),
                emptylist))
   ```

   debe resultar un valor expresado que representa la lista `(4 (3))`.
5. Agrega una operación `list` al lenguaje. Esta operación debe tomar cualquier cantidad de argumentos y regresar un valor expresado de la lista de sus valores. Por ejemplo,

[a] eduardo.acuna@unison.mx

```
let x = 4
in list(x, -(x,1), -(x,3))
```

debe resultar en un valor expresado que representa a la lista `(4 3 1)`.

6. En un lenguaje real, uno pudiera querer muchos operadores como los que incorporamos en los ejercicios pasados. Adecua el código del intérprete de LET para que sea simple agregar nuevos operadores.
7. Incorpora al lenguaje expresiones `cond`. Usa la gramática

$$\textit{Expression} \longrightarrow \textbf{cond}\ \{\textit{Expression}\ \textbf{=>}\ \textit{Expression}\}^*\ \textbf{end}$$

En esta expresión, las expresiones de los lados izquierdos de los **=>** son evaluadas en orden hasta que una de ellas regresa un valor verdadero. Entonces el valor de toda la expresión es el valor de la expresión correspondiente al lado derecho de esa **=>**. Si ninguno de los lados izquierdos es verdadero, la expresión debe reportar un error.

8. Cambia los valores del lenguaje para que los enteros sean los únicos valores expresados. Modifica `if` para que el valor de 0 sea tratado como falso y todos los otros valores sean tratados como verdaderos. Modifica los predicados de manera consistente.
9. Como una alternativa al ejercicio anterior, agrega una nueva categoría sintáctica *Bool-exp* de expresiones booleanas al lenguaje. Cambia la producción para expresiones condicionales para que sea

$$\textit{Expression} \longrightarrow \textbf{if}\ \textit{Bool-exp}\ \textbf{then}\ \textit{Expression}\ \textbf{else}\ \textit{Expression}$$

Escribe producciones apropiadas para *Bool-exp* y especifica su semántica con `value-of-bool-exp` (puedes abreviarlo como $\mathcal{B}$). ¿En dónde terminan estando los predicados del ejercicio 3 con este cambio?

10. Modifica la implementación del intérprete agregando una nueva operación `print` que toma un argumento, lo imprime, y regresa el entero 1. ¿Por qué esta operación no es expresable en nuestro método de especificación formal?
11. Extiende el lenguaje para que las expresiones `let` puedan vincular una cantidad arbitraria de variables, usando la producción,

$$\textit{Expression} \longrightarrow \textbf{let}\ \{\textit{Identifier}\ \textbf{=}\ \textit{Expression}\}^*\ \textbf{in}\ \textit{Expression}$$

Al igual que el `let` de Racket, cada uno de los lados derechos es evaluado en el entorno y el cuerpo del `let` es evaluado con cada nueva variable vinculada al valor de su lado derecho asociado. Por ejemplo,

```
let x = 30
in let x = -(x,1)
       y = -(x,2)
   in -(x,y)
```

debe ser evaluada a 1.

12. Extiende el lenguaje con una expresión `let*` que funciona como en Racket, de tal manera que,

```
let x = 30
in let* x = -(x,1) y = -(x,2)
   in -(x,y)
```

sea evaluada a 2.

13. Agrega una expresión al lenguaje de acuerdo a la siguiente regla,

$$\textit{Expression} \longrightarrow \textbf{unpack}\ \{\textit{Identifier}\}^*\ \textbf{=}\ \textit{Expression}\ \textbf{in}\ \textit{Expression}$$

tal que `unpack x y z = lst in ...` vincula `x`, `y` y `z` a los elementos de `lst` si `lst` es una lista con exactamente tres elementos, reportando un error en otro caso. Por ejemplo,

```
let u = 7
in unpack x y = cons(u, cons(3, emptylist))
   in -(x,y)
```

debe ser evaluada a 4.

14. Consideremos funciones de compilación `compile-program` (para compilar programas, abreviada $C_{\mathcal{P}}$) y `compile` (para compilar expresiones, abreviada $C_{\mathcal{E}}$) que toman de argumento un árbol de sintaxis y regresan un árbol de sintaxis. Implementa estas funciones para que el lenguaje compile el `cond` del ejercicio 7, la extensión a `let` del ejercicio 11, y el `let*` del ejercicio 12, utilizando el lenguaje LET base y las extensiones de los ejercicios 4, 5 y 13.