`calc` en Ruby

Juan Francisco Cardona McCormick

12/11/2015

1. Introducción

Durante una parte del semestre hemos aprendido programación orientada a objetos con el lenguaje de programación Ruby construyendo un interpretador de expresiones con memoria. Ésta práctica, continuará ampliando este proyecto para aprender otros aspectos del lenguaje de programación Ruby, como entrada y salida, colecciones y otros. En este documento se propondrán los cambios que se harán para la el proyecto calc.

2. Definición

Son varias las modificaciones que vamos hacer al proyecto:

- Una gramática que permite recibir más de una expresión.
- Inicialización de identificadores de variables y el uso de variables dentro de las expresiones.
- Ampliación de las funciones de la memoria de la calculadora.

Aunque son varias modificaciones al proyecto original, esto no implica que vamos a tener varios programas: uno para la calculador original, otro para las lectura interactiva de varias expresiones, etc. ***Se va a generar un único ejecutable*** `calc`[1] (mirar las secciones: 2.1.5, 2.1.1 y **??**).

[1] No es necesario poner la extensión `.exe` por que se sobrentiende que es un ejecutable del lenguaje de Ruby.

2.1. Gramática modificada

La siguiente es la nueva versión de la gramática independiente de contexto $LL(1)$:

$$
\begin{array}{lcl}
\mathit{Prog} & \rightarrow & \mathit{ListExpr} \\
\mathit{ListExpr} & \rightarrow & \mathit{Expr}\ \text{EOL}\ \mathit{ListExpr}\ \mid\ \text{EOF} \\
\mathit{Expr} & \rightarrow & \mathit{Term}\ \mathit{RestExpr} \\
\mathit{RestExpr} & \rightarrow & +\ \mathit{Term}\ \mathit{RestExpr}\ \mid\ -\ \mathit{Term}\ \mathit{RestExpr}\ \mid\ \epsilon \\
\mathit{Term} & \rightarrow & \mathit{Storable}\ \mathit{RestTerm} \\
\mathit{RestTerm} & \rightarrow & *\ \mathit{Term}\ \mathit{RestTerm}\ \mid\ /\ \mathit{Term}\ \mathit{RestTerm} \\
 & \mid & \%\ \mathit{Term}\ \mathit{RestTerm}\ \mid\ \epsilon \\
\mathit{Storable} & \rightarrow & \mathit{Factor}\ \mathit{MemOperation} \\
\mathit{MemOperation} & \rightarrow & \text{S}\ \mid\ \text{P}\ \mid\ \text{M}\ \mid\ \epsilon \\
\mathit{Factor} & \rightarrow & \textbf{number}\ \mid\ \mathit{Assignable}\ \mid\ \text{R}\ \mid\ \text{C}\ \mid\ (\ \mathit{Expr}\ ) \\
\mathit{Assignable} & \rightarrow & \textbf{identifier}\ \mathit{Assign} \\
\mathit{Assign} & \rightarrow & =\ \mathit{Expr}\ \mid \epsilon
\end{array}
$$

Esta gramática permite manejar una lista de expresiones y asignación de variables:

```
$ ./calc
> 3 + 4 <eol>
=> 7
> 8 + 9 <eol>
=> 17
> 8 + <eol>
* parse error
> 3 - 2 <eol>
=> 1
> a = 10 + R
=> a <- 10
> <eof>
$
```

Se observa que el programa ya no envía el mensaje de solicitar una expresión, sino que muestra lo que se conoce como un *prompt*[2]. Este prompt

[2]Muchas de las expresiones que vamos a utilizar son dadas en inglés por que en español

indica al usuario que entre una expresión y el programa responderá con => para indicar el resultado o $*$ para indicar un error. El usuario debe entrar una sesión con una línea nueva < `eol` >, esta línea nueva es simplemente presionar la tecla *return*[3]. Finalmente, el usuario puede dejar de entrar expresiones al indicar un fin de fichero (*eof - end of file*). El usuario puede indicar el fin de un fichero utilizando en linux (cygwin o Mac) la combinación de las teclas `Ctrl+D` y en Windows[4] con la combinación de teclas `Ctrl+Z`. Con el fin de fichero el programa `calc` termina su ejecución y vuelve el prompt de la terminal para esperar otro comando.

2.1.1. Sintaxis línea de comandos

La siguiente es la sintaxis de la línea de comandos:

$$
\begin{array}{lcl}
Comandos & \rightarrow & Options\ [Fichero]\ \ldots \\
Options & \rightarrow & [PrefEnv]\ [InitVar]\ \ldots \\
PrefEnv & \rightarrow & \text{-e} \\
InitVar & \rightarrow & \text{-v "}\textbf{identifier} = \text{number"} \\
Fichero & \rightarrow & \textbf{filename}
\end{array}
$$

Una línea de comando esta formada de dos partes: Opciones y una lista de ficheros. Las opciones puede tener dos partes la primera establece si las variables de ambiente tiene preferencia sobre las variables definidas en la línea de comandos y la segunda establece una lista posiblemente vacía de la definición de variables.

La parte final de una línea de comandos viene con una lista posiblemente vacía de ficheros donde se encuentra las instrucciones en el lenguaje de la calculador que serán ejecutadas por la misma.

---

no existe una traducción adecuada. En este caso *prompt* tiene traducción literal en inglés de solicitar a alguien de decir o hacer algo, en este caso > es una forma que la calculadora le indica al usuario que digite una expresión.

[3]Esto depende de la plataforma en Linux, Unix y Mac es simplemente un solo carácter \n, en Windows es la combinación de dos caracteres \r\n. El programa debe ser independiente de la plataforma

[4]Decida explorar utilizar otro compilador como pro ejemplo el compilador de Visual Studio

2.1.2. Inicialización de variables ambiente

Las variables puede ser inicializadas desde variables del ambiente. Las variables inician con el prefijo `CALCVAR` y continúa con el nombre de la variable y luego el valor que está tendrá:

```
$ export CALCVARa=10
$ export CALCVARb=20
$ export CALCVARUnaVar=30
$ calc
> val = a + b + UnaVar
=> val <- 60
> <eof>
```

2.1.3. Inicialización de variables

Las variables (o identificadores) de una expresión de consideran que tiene valores cero:

```
$ ./calc
> 3 + b <eol>
=> 3
> <eof>
$
```

La gramática permitirá la inicialización de variables, a través de la línea de comandos:

```
$ ./calc -v b=10 -v c=20
> 3 + b <eol>
=> 13
> 2 * c <eol>
=> 40
> <eof>
$
```

La gramática permitirá la inicialización de variables, a través de la asignación:

```
$ ./calc
> a = 10
=> a <- 10
> b = 20
=> b <- 10
> c = a * b
=> c <- 200
> <eof>
```

Las variables también puede tener nombres largos:

```
$ ./calc -v length=20 -v height=10
> area = length * height
=> area <- 200
> <eof>
```

2.1.4. Preferencia en inicialización de varibles

Las variables puede ser inicializadas de dos formas: variables de ambiente o línea de comandos. ¿Qué pasa cuando una variable es inicializada en ambos lados? Normalmente, se tiene preferencia de las variables definidas en la línea de comandos:

```
$ export CALCVARlength=10
$ export CALCVARh=20
$ ./calc -v length=30 -v h=40
> area = length * h
=> area <- 1200
```

Pero esta preferencia se puede cambiar sí la línea de comandos se le añade que se prefiere las variables de ambiente `-e`:

```
$ export CALCVARlength=10
$ export CALCVARh=20
$ ./calc -e -v length=30 -v h=40
> area = length * h
=> area <- 200
```

Si preferencia está en las variables de ambiente, pero está no esta definida en las variables de ambiente, se prefiere la definición en la línea de comandos.

```
$ export CALCVARlength=10
$ ./calc -e -v length=30 -v h=40
> area = length * h
=> area <- 400
```

Si no están definidas completamente su valor por omisión será 0:

```
$ export CALCVARlength=10
$ ./calc -e
> area = length + h
=> area <- 10
```

2.1.5. Lectura de más de un fichero

La entrada de las expresiones se puede cambiar si estas son leídas directamente de uno o más ficheros. Suponga que tenemos dos ficheros con extensión `.calc` para indicar que son programas que contiene expresiones aritméticas:

```
$ cat expresion1.calc
1 + b
2 * 4S + R
(3 + 4) * d
val = R * 10
<eof>
```

```
$ cat expresion2.calc
1S + 4M + R
a + c + d
val = val - 10
<eof>
```

Para ejecutar `calc` con fichero en la línea de comando se indica al final el nombre del fichero que se quiere procesar:

```
$ ./calc expresion1.calc
=> 1
=> 12
=> 0
=> val <- 40
```

Se puede procesar con más de un fichero:

```
$ ./calc expresion1.calc expresion2.calc
=> 1
=> 12
=> 0
=> val <- 40
=> -5
=> 0
=> val <- 30
```

También se puede combinar con la inicialización de la variables:

```
$ ./calc -v a=1 -v b=2 -v c=3 -v d=4 expresion1.calc expresion2.calc
=> 3
=> 12
=> 28
=> val <- 40
=> -5
=> 8
=> val <- 30
```

2.1.6. Asignaciones

La asignación es una operación que se puede hacer varias veces dentro de una expresión, puesto que la gramática así lo permite. En el siguiente ejemplo se ve como esto se puede hacer:

```
$ ./calc
> (a = 10) + (b = 20) + (c= 30)
=> 60 [a <- 10, b <- 20, c <- 30]
```

La gramática también permite anidar las asignaciones:

```
$ ./calc
> (a = (b = (c = 30))) + (a = 20)
=> 50 [c <- 30, b <- 30, a <- 30, a <- 20]
```

2.2. Implementación

En la implementación actual de Ruby se debe implementar completamente la gramática, es decir deben estar implementas las funciones para `Prog`, `ListExpr` y las restantes.

Se va hacer revisión de la implementación completa de la gramática.

3. Requerimientos administrativos

Trabajo individual: El trabajo para esta práctica es individual. *Cada* estudiante ya tiene un repositorio en (`http://riouxsvn.com`) con un usuario con el mismo nombre del usuario asignado en la universidad (<usuario>) y un directorio `proyecto/rubycalc`. Deben invitar al profesor (usuario: $fcardona$ en riouxsvn) y al monitor correspondiente: Santiago Montoya (usuario: *santiagom* en riouxsvn) o Santiago Suarez (usuario: $ssuarez6$ en rioxsvn) o Cristian Ospina (usuario: $cospin18$ en riousvn).

Primera entrega: Viernes 27 de Noviembre hasta las 06:00 am. Esta se hará automática a través del sistema de control de versiones. Se *debe* cumplir todos los requerimientos señalados en este documento.

Sustentación: El día antes de la entrega de la práctica, se les enviará un enlace URL a quienes deberán presentar la sustentación donde deben registrar la misma. Las sustentaciones se harán a partir del viernes 27 de Noviembre en horarios que se envían en el enlace. No sustentarán todos aproximadamente la mitad de lo estudiantes registrados en dicha fecha se les enviará un correo confirmando la presentación de la sustentación de la primera práctica.

Pruebas: El proyecto se copiará con una serie de pruebas que consiste en un conjunto de expresiones: 20 en total. Las mismas pruebas para la práctica de C++ se utilizarán para la evaluar la práctica de Ruby

Nota práctica: Se calcula con la siguiente formula para quienes presenten la sustentación:

$$Nota\ segunda\ práctica = Sustentación \times Impl \times (5 \times \sum_{j=1}^{5} (\frac{1}{5} * (\frac{(\sum_{i=1}^{20} prueba_i^j)}{20})))$$

Donde $Sustentación$ es valor obtenido en la sustentación ($0 \leq Sustentación \leq 1$); $prueba_i^j$ indica si la prueba i pasó o falló en el caso j ($0 \leq prueba_i^j \leq 1$). Donde $Impl$ ($0 \leq Impl \leq 1$) es el nivel de implementación.

La siguiente fórmula aplica para quienes no hicieron la primera sustentación por que no fueron formalmente invitados:

$$Nota\ segunda\ práctica = Impl \times 5 \times \sum_{j=1}^{3}(\frac{1}{5} * (\frac{(\sum_{i=1}^{20} prueba_i^j)}{20}))$$

Control de versiones: El control de versiones no es solamente un herramienta que facilite la comunicación entre los miembros del grupo y del control de versiones, sino que también ayudará al profesor a llevar un control sobre el desarrollo de la práctica. Se espera que las diferentes registros dentro del control de versiones sean cambios graduales. En caso contrario, se procederá a realizar un escutrinio a fondo del manejo de control de versiones para evitar fraudes.

Fraudes: Seguiremos los lineamientos establecidos por la universidad con respecto a las conductas que atentan contra el orden académico. En el suguiente enlace `http://www.eafit.edu.co/institucional/reglamentos/Documents/pregrado/regimen-disciplinario/cap1.pdf` se encuentra la parte del régimen disciplinario de la universidad.