



UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

## Lenguajes de Programacion Examen Parcial I



■ Edgar Montiel Ledesma  
317317794

■ Carlos Daniel Cortes Jimenez  
420004846

1. En el lenguaje de programacion Zoo, los nombres para variables deben empezar con el caracter 'V' seguido por una cadena cualquiera no vacia de caracteres 'o' o 'z'.

a) Defina un juicio ozv tal que s ozv se cumpla si y solo si s es un nombre valido de variable en Zoo.

$ozv(s) \Leftrightarrow (s \text{ comienza con } V) \wedge (s \text{ contiene al menos un } o \vee z \text{ despues de } V)$

Donde:

- s representa la cadena que se esta evaluando.
- $\Leftrightarrow$  "si y solo si"
- (s comienza con V) es verdadero si el primer carácter de la cadena s es V.
- (s contiene al menos un o  $\vee$  z despues de V) es verdadero si la cadena s contiene al menos un carácter o  $\vee$  z después del primer caracter V.

Teniendo la siginete gramatica

$E ::= Vs$

$s ::= os \mid zs \mid o \mid z$

$$\frac{\frac{P}{Q}(\text{Premisa}) \quad \frac{Q \Rightarrow R}{R}(\text{Premisa})}{\frac{P}{R}(\text{Modus Ponens})}$$

b) Derive V ozo ozv usando sus reglas.

c) Enuncie el principio de induccion estructural para el juicio ozv y utilicelo para demostrar que: si w ozv entonces  $\exists u \in o, z + (w = Vu)$

Si  $w = "V"$ , entonces  $\exists u \in o, z (w = Vu)$ .

Paso de la inducción: Supongamos que tenemos una cadena  $w$  que es un nombre válido de variable en el lenguaje Zoo, es decir,  $ozv(w)$  es verdadero. Queremos demostrar que para cualquier cadena más compleja  $w'$  que se pueda formar al agregar un carácter ó al final de  $w$ , la propiedad sigue siendo verdadera. Entonces, consideremos dos casos:

Caso 1: Si agregamos al final de  $w$  para formar  $wz$ ; entonces podemos decir que  $wz \neq w$  ó: Dado que  $w$  ya es un nombre válido de variable, podemos tomar  $u$  igual a  $w$  y todavía se cumplirá la propiedad:

Si  $wz \neq w$  ó; entonces  $\exists u \in o, z (wz \neq V u)$ .

Caso 2: Podemos decir que  $w' = w \cdot z$  si agregamos  $z$  al final de  $w$  para formar  $w'$ . Dado que  $w$  ya es un nombre válido de variable, podemos tomar  $u$  igual a  $w$  y aún se cumplirá la propiedad:

Si  $wz \neq w$  ó; entonces  $\exists u \in o, z (wz \neq V u)$ .

En ambos casos demostramos que si la propiedad es verdadera para  $w$ , también es verdadera para  $w'$ , donde  $w'$  se construye agregando  $z$  al final de  $w$ .

Por lo tanto la propiedad es verdadera para todas las cadenas  $w$ , que son nombres de variable válidos en el lenguaje Zoo.

2. En muchos lenguajes de programación las expresiones flotantes incluyen el uso de notación científica. Por ejemplo en Pascal la notación científica se expresa usando una letra  $e$ . Por ejemplo  $+9.67e-15$  significa  $9.67 \times 10^{-15}$  y las expresiones flotantes son de alguna de las siguientes tres formas:

`s.u s e r s . u e r`

donde  $s, r$  son enteros signados y  $u$  es un entero no signado.

- a) Defina un juicio `pfloat` que genere a las expresiones flotantes de Pascal. Observe que debe también definir juicios para enteros signados y no signados.
- b) Utilice su definición para dar una derivación de  $+9.67e-15$ .

Sugerencia: siempre es útil definir primero una gramática libre de contexto adecuada y transformarla en el juicio deseado.

3. Un proceso estándar en la implementación de compiladores es el llamado plegado de constantes (*constant folding*) que consiste en reconocer y evaluar expresiones constantes en tiempo de compilación en vez de computar los resultados en tiempo de ejecución, esto incluye simplificaciones usando propiedades aritméticas. Por ejemplo la expresión  $(2 + 3 + y) * (x + 4 * 5)$  se simplifica a  $(5 + y) * (x + 20)$ , la expresión  $2 * x + 0$  se simplifica en  $2 * x$  y la expresión  $1 * (x + 2 * z)$  se simplifica en  $x + 2 * z$ . Considere el siguiente lenguaje EAs de expresiones aritméticas simples.

$e ::= x \mid n \mid e + e \mid - e \mid e * e \mid (e)$

- a) Defina una función `cfold :: EAs → EAs` que realice el proceso de plegado de constantes. Puede utilizar pseudocódigo de Haskell.

```

cfold :: EAs -> EAs
cfold (x) = x
cfold (n) = n
cfold (e1 + e2) | isNumber e1 && isNumber e2 = e1 + e2
                  | otherwise = cfold e1 + cfold e2
cfold (-e) | isNumber e = -e
             | otherwise = -cfold e
cfold (e1 * e2) | isNumber e1 && isNumber e2 = e1 * e2
                  | otherwise = cfold e1 * cfold e2
cfold (e) = e

isNumber :: EAs -> Bool
isNumber (n) = True
isNumber (-) = False

```

- b) Verifique que su definicion es correcta mediante el computo de  $cfold((5 + 2) + x * 1)$

Verificamos que la definicion de  $cfold$  es correcta:

Evaluablemos primero  $cfold((5 + 2) \rightarrow cfold(7))$  dado que 7 es una constante se simplifica a 7.

Ahora eevaluablemos  $cfold(x * 1) \rightarrow cfold(x)$  ya que 1 es una constante, y al multiplicar por 1 obtendremos el mismo valor por lo que la expresi3n se simplifica a solo  $x$ .

Por lo que el resultado de aplicar  $cfold$  a  $(5 + 2) + x * 1$  es  $7 + x$ .

- c) Defina el interprete denotativo para EAs y demuestre su correccion con respecto al plegado de constantes, es decir, demuestre que para cualquier expresion  $e$  se cumple que

$$evalse = evals(cfolde)$$

```

eval :: EVar -> EAs -> Integer
eval s (x) = s x
eval s (n) = n
eval s (e1 + e2) = eval s e1 + eval s e2
eval s (-e) = -eval s e
eval s (e1 * e2) = eval s e1 * eval s e2

```

Ahora demostramos la correcci3n del int3rprete con respecto al plegado de constantes.

**Demostraci3n:** Induccion estructural sobre la expresi3n  $e$ .

**Casos Base:**

- Si  $e$  es una variable  $x$ , en ambos lados de la expresi3n son iguales, en  $eval\ s\ e$  devuelve el valor esperado  $x$ , as3 como en  $eval\ s\ (cfold\ e)$  tambien devuelve  $x$ , ya que  $cfold$  no hace ninguna simplificaci3n. Por lo tanto  $eval\ s\ e = eval\ s\ (cfold\ e)$ .

- Si  $e$  es una constante  $n$ , así como en el caso anterior ambas partes serán iguales, debido a que  $cfold$  no altera constantes, por lo que en ambos lados devuelve  $n$ , entonces  $eval\ s\ e = eval\ s\ (cfold\ e)$ .

**Hipótesis de Inducción(H.I.):** Para una expresión  $e$  arbitraria y estado  $s$ , se cumple que  $eval\ s\ e = eval\ s\ (cfold\ e)$ .

**Paso Inductivo:**

- Suma(+): Supongamos que se cumple la H.I. para  $e1$  y  $e2$ , esto es  $eval\ s\ e1 = eval\ s\ (cfold\ e1)$  y  $eval\ s\ e2 = eval\ s\ (cfold\ e2)$ . Consideremos  $e = e1 + e2$ , para este caso  $eval\ s\ (e1 + e2)$  realiza la suma de los valores  $e1$  y  $e2$  que sería  $eval\ s\ e1 + eval\ s\ e2$ . Por lo que se tiene  $eval\ s\ (e1 + e2) = eval\ s\ e1 + eval\ s\ e2$ . Hacemos lo mismo para  $eval\ s\ (cfold\ e1 + cfold\ e2)$ , el cual realiza la suma de los valores  $cfold\ e1$  y  $cfold\ e2$  que sería  $eval\ s\ (cfold\ e1) + eval\ s\ (cfold\ e2)$ . Por H.I. tenemos que  $eval\ s\ (cfold\ e1) = eval\ s\ e1$ , a su vez también se tiene que  $eval\ s\ (cfold\ e2) = eval\ s\ e2$ . Por lo tanto  $eval\ s\ (cfold\ e1 + cfold\ e2) = eval\ s\ e1 + eval\ s\ e2$ . Se demuestra que  $eval\ s\ (e1 + e2) = eval\ s\ (cfold\ e1 + cfold\ e2)$ .
- Multiplicación(\*): Analogamente al caso de la suma, supongamos que se cumple la H.I. para  $e1$  y  $e2$ , esto es  $eval\ s\ e1 = eval\ s\ (cfold\ e1)$  y  $eval\ s\ e2 = eval\ s\ (cfold\ e2)$ . Consideremos  $e = e1 * e2$ , para este caso  $eval\ s\ (e1 * e2)$  realiza el producto de los valores  $e1$  y  $e2$  que sería  $eval\ s\ e1 * eval\ s\ e2$ . Por lo que se tiene  $eval\ s\ (e1 * e2) = eval\ s\ e1 * eval\ s\ e2$ . Hacemos lo mismo para  $eval\ s\ (cfold\ e1 * cfold\ e2)$ , el cual realiza el producto de los valores  $cfold\ e1$  y  $cfold\ e2$  que sería  $eval\ s\ (cfold\ e1) * eval\ s\ (cfold\ e2)$ . Por H.I. tenemos que  $eval\ s\ (cfold\ e1) = eval\ s\ e1$ , a su vez también se tiene que  $eval\ s\ (cfold\ e2) = eval\ s\ e2$ . Por lo tanto  $eval\ s\ (cfold\ e1 * cfold\ e2) = eval\ s\ e1 * eval\ s\ e2$ . Se demuestra que  $eval\ s\ (e1 * e2) = eval\ s\ (cfold\ e1 * cfold\ e2)$  y entonces  $eval\ s\ e = eval\ s\ (cfold\ e)$ .
- Negación(-): Supongamos que se cumple la H.I. para  $e$ , entonces se tiene que  $eval\ s\ e = eval\ s\ (cfold\ e)$ . Consideremos  $e = -e$ , para este caso  $eval\ s\ (-e)$  realiza la negación del valor  $e$  que sería  $-eval\ s\ e$ . Por lo que se tiene  $eval\ s\ (-e) = -eval\ s\ e$ . Hacemos lo mismo para  $eval\ s\ (-cfold\ e)$  el cual realiza la negación del valor  $cfold\ e$  que sería  $-eval\ s\ (cfold\ e)$ . Por H.I. tenemos que  $eval\ s\ (cfold\ e) = eval\ s\ e$ . Por lo tanto  $eval\ s\ (-cfold\ e) = -eval\ s\ e$ . Se demuestra que  $eval\ s\ (-e) = eval\ s\ (-cfold\ e)$ .

Por lo tanto, para cualquier expresión  $e$  se cumple que  $eval\ s\ e = eval\ s\ (cfold\ e)$  en el lenguaje EAs.

4. A continuación se define la sintaxis concreta de un lenguaje funcional muy simple.

$$e ::= x | n | e1\ e2 | fun(x) \rightarrow e$$

En donde el primer constructor representa las variables del lenguaje, el segundo números naturales, el tercero aplicación de función y el último la definición de funciones.

- a) Traduce la gramática anterior a una definición inductiva con reglas de inferencia.
- b) Diseña una sintaxis abstracta apropiada para este lenguaje. Hint: Primero observa si es necesario alguna especie de ligando como el que define el operador `let` visto en clase.
- c) Escribe las reglas para la relación de análisis sintáctico ( $\leftrightarrow$ ) del lenguaje.

- d) Diseña un algoritmo de sustitución para este lenguaje.
- e) La relación de  $\alpha$ -equivalencia en este lenguaje se da respecto al operador de definición de función  $\text{fun}$ , se dice que dos expresiones son  $\alpha$ -equivalentes si solo difieren en el nombre de la variable del parámetro de la función. Por ejemplo:

$$\text{fun}(x) \rightarrow x \equiv \alpha \text{fun}(y) \rightarrow y$$

Demuestra que  $\equiv \alpha$  es una relación de equivalencia.

5. Juanito Alimana quiere definir un lenguaje que le permita controlar un robot con movimientos y funcionalidades muy simples. El robot se mueve sobre una cuadrícula siguiendo las instrucciones especificadas por el programa. Al inicio el robot se encuentra en la coordenada  $(0, 0)$  y viendo hacia el norte. El programa consiste en una secuencia posiblemente vacía de los comandos  $\text{move}$  y  $\text{turn}$  separados por punto y coma, cada comando tiene el siguiente funcionamiento:

- $\text{turn}$  hace que el robot de un giro de 90 grados en el sentido de las manecillas del reloj.
- $\text{move}$  provoca que el robot avance una casilla en la dirección hacia la que está viendo.

Un ejemplo de un programa válido es:

`move;turn;move;turn;turn;turn;move`

Al final del programa el robot termina en la casilla  $(2, 1)$ . La primera entrada de la coordenada indica la posición vertical mientras que la segunda es la posición horizontal.

Juanito nos pidió definir la semántica operacional de paso pequeño para el lenguaje que controla al robot. Para esto responde las siguientes preguntas formalmente, para diseñar un sistema de transición:

- a) Determina el conjunto de estados.
- b) Identifica los estados iniciales y finales del sistema de transición.
- c) Define la función de transición  $\rightarrow_R$  que indique cómo se debe transitar entre los estados del sistema. De tal forma que defina una semántica operacional de paso pequeño.
- d) Muestra paso a paso la ejecución del programa

`move;turn;move;turn;turn;turn;move`

usando la relación  $\rightarrow_R$  y partiendo del estado inicial correspondiente.