Universidad Nacional de Rosario
Instituto Politécnico Superior
Analista Universitario en Sistemas
Programacion Profesional

Trabajo Práctico

1 Introducción

Se presenta un lenguaje imperativo simple con variables enteras y comandos para asignación, composición secuencial, ejecución condicional (**if**) y ciclos (**while**). Se epecifica su sintaxis abstracta, su sintaxis concreta, una realización de su sintaxis abstracta en *Haskell*, y por úlitmo su semántica operacional estructural —*small-step semantics*. El objetivo del trabajo es construir un intérprete en Haskell para el lenguaje presentado.

El trabajo se puede realizar individualmente o en grupos de hasta 3 personas.

- En papel, un informe con los ejercicios resueltos sin omitir todo el código que haya escrito
- En formato digital, el código fuente del intérprete (archivos `Eval.hs`) y del parser (archivo .hs).

2 Especificación del Lenguaje Imperativo Simple (LIS)

2.1 Sintaxis Abstracta

Aunque es posible especificar la semántica de un lenguaje como una función sobre el conjunto de cadenas de caracteres de su sintaxis concreta, una especificación de ese estilo es innecesariamente complicada. Las frases de un lenguaje formal que se representan como cadenas de caracteres son en realidad entidades abstractas y es mucho más conveniente definir la semántica del lenguaje sobre estas entidades. La *sintaxis abstracta* de un lenguaje formal es la especificación de los conjuntos de frases abstractas del lenguaje.

Por otro lado, aunque las frases sean conceptualmente abstractas, se necesita alguna notación para representarlas. Una sintaxis abstracta se puede expresar utilizando una *gramática abstracta*, la cual define conjuntos de frases independientes de cualquier representación particular, pero al mismo tiempo provee una notación simple para estas frases. Una gramática abstracta para LIS es la siguiente:

$$
\begin{aligned}
\langle intexp \rangle ::= \; & \langle nat \rangle \\
| \; & \langle var \rangle \\
| \; & -_u \langle intexp \rangle \\
| \; & \langle intexp \rangle + \langle intexp \rangle \\
| \; & \langle intexp \rangle -_b \langle intexp \rangle \\
| \; & \langle intexp \rangle \times \langle intexp \rangle \\
| \; & \langle intexp \rangle \div \langle intexp \rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle boolexp \rangle ::= \; & \textbf{true} \mid \textbf{false} \\
| \; & \langle intexp \rangle = \langle intexp \rangle \\
| \; & \langle intexp \rangle < \langle intexp \rangle \\
| \; & \langle intexp \rangle > \langle intexp \rangle \\
| \; & \langle boolexp \rangle \wedge \langle boolexp \rangle \\
| \; & \langle boolexp \rangle \vee \langle boolexp \rangle \\
| \; & \neg \langle boolexp \rangle
\end{aligned}
$$

⟨*comm*⟩ ::= **skip**
| ⟨*var*⟩ := ⟨*intexp*⟩
| ⟨*comm*⟩ ; ⟨*comm*⟩
| **if** ⟨*boolexp*⟩ **then** ⟨*comm*⟩ **else** ⟨*comm*⟩
| **while** ⟨*boolexp*⟩ **do** ⟨*comm*⟩

donde ⟨*var*⟩ representa al conjunto de identificadores de variables y ⟨*nat*⟩ al conjunto de los números naturales.

2.2 Sintaxis Concreta

La sintaxis concreta de un lenguaje incluye todas las características que se observan en un progama fuente, como delimitadores y paréntesis. La sintaxis concreta de LIS se describe por la siguiente gramática libre de contexto en BNF:

⟨*digit*⟩ ::= '`0`' | '`1`' | ... | '`9`'

⟨*letter*⟩ ::= '`a`' | ... | '`Z`'

⟨*nat*⟩ ::= ⟨*digit*⟩ | ⟨*digit*⟩⟨*nat*⟩

⟨*var*⟩ ::= ⟨*letter*⟩ | ⟨*letter*⟩⟨*var*⟩

⟨*intexp*⟩ ::= ⟨*nat*⟩
| ⟨*var*⟩
| '`-`' ⟨*intexp*⟩
| ⟨*intexp*⟩ '`+`' ⟨*intexp*⟩
| ⟨*intexp*⟩ '`-`' ⟨*intexp*⟩
| ⟨*intexp*⟩ '`*`' ⟨*intexp*⟩
| ⟨*intexp*⟩ '`/`' ⟨*intexp*⟩
| '`(`' ⟨*intexp*⟩ '`)`'

⟨*boolexp*⟩ ::= '`true`' | '`false`'
| ⟨*intexp*⟩ '`=`' ⟨*intexp*⟩
| ⟨*intexp*⟩ '`<`' ⟨*intexp*⟩
| ⟨*intexp*⟩ '`>`' ⟨*intexp*⟩
| ⟨*boolexp*⟩ '`&`' ⟨*boolexp*⟩
| ⟨*boolexp*⟩ '`|`' ⟨*boolexp*⟩
| '`~`' ⟨*boolexp*⟩
| '`(`' ⟨*boolexp*⟩ '`)`'

⟨*comm*⟩ ::= '`skip`'
| ⟨*var*⟩ '`:=`' ⟨*intexp*⟩
| ⟨*comm*⟩ '`;`' ⟨*comm*⟩
| '`if`' ⟨*boolexp*⟩ '`then`' ⟨*comm*⟩ '`else`' ⟨*comm*⟩ '`end`'
| '`while`' ⟨*boolexp*⟩ '`do`' ⟨*comm*⟩ '`end`'

La gramática así definida es ambigüa. Para desambigüarla, se conviene una *lista de precedencia* para los operadores del lenguaje, enumerándolos en grupos de orden decreciente de precedencia:

$$-_u \quad (\times \div) \quad (+ -_b) \quad (= < >) \quad \neg \quad \wedge \quad \vee \quad := \quad ;$$

donde todos los operadores binarios asocian a izquierda excepto =, < y > que no son asociativos (la asociatividad es irrelevante para :=, ya que ni $(x_0 := x_1) := x_2$ ni $x_0 := (x_1 := x_2)$ satisfacen la gramática)

Ejercicio 2.2.1 *Extienda las sintaxis abstracta y concreta de LIS para incluir una nueva expresión entera, al estilo del operador condicional ternario " `?:` " del lenguaje C*

2.3 Realización de la Sintaxis Abstracta en Haskell

Cada no terminal de la gramática de la sintaxis abstracta puede representarse como un tipo de datos; cada regla de la forma

$$L ::= s_0 R_0 s_1 R_1 \ldots R_{n-1} s_n$$

donde $s_0, \ldots, s_n$ son secuencias de símbolos terminales, da lugar a un constructor de tipo

$$R_0 \times R_1 \times \ldots \times R_{n-1} \to L$$

```
-- Identificadores de Variable
type Variable = String

-- Expresiones Aritmeticas
data IntExp = Const Int
            | Var Variable
            | UMinus IntExp
            | Plus IntExp IntExp
            | Minus IntExp IntExp
            | Times IntExp IntExp
            | Div IntExp IntExp

-- Expresiones Booleanas
data BoolExp = BTrue
             | BFalse
             | Eq IntExp IntExp
             | Lt IntExp IntExp
             | Gt IntExp IntExp
             | And BoolExp BoolExp
             | Or BoolExp BoolExp
             | Not BoolExp

-- Comandos
-- Observar que solo se permiten variables de un tipo (entero)
data Comm = Skip
          | Let Variable IntExp
          | Seq Comm Comm
          | Cond BoolExp Comm Comm
          | While BoolExp Comm
```

Ejercicio 2.3.1 *Extienda la realización de la sintaxis abstracta en Haskell para incluir el operador ternario descripto en el Ejercicio 2.2.1*

Ejercicio 2.3.2 *Implementar un parser en el archivo* `Parser.hs`*, que traduzca un programa LIS en su representación concreta a un árbol de sintaxis abstracta utilizando la biblioteca Parsec.*

Parsec es una biblioteca para construir parsers con combinadores similares a los vistos en clase, pero mucho más potente (`https://hackage.haskell.org/package/parsec`). Además de permitir trabajar con combinadores a nivel de caracter, `Parsec` permite trabajar con `tokens`. Es decir que el parseo se hace en dos pasos:

a) Se transforma la cadena de entrada en una lista de tokens. Cada token indica si se tiene un identificador, una palabra clave, un operador, etc. Durante esta transformación se eliminan espacios y comentarios. Se utiliza la función `makeTokenParser` para generar parsers que funcionen sobre tokens. Para ello, se especifica la forma de los comentarios, identificadores, etc. En particular, en `Parser.hs`, en la definición lis, se han configurado las palabras clave y los nombres de los operadores (con las del lenguaje LIS), y el formato de los comentarios (tomando los delimitadores $/*$ y $*/$ para bloques y // para comentarios en ĺnea, como en el lenguaje $C++$ o *Java*). El uso de un parser de tokens hace que no sea necesario lidiar con espacios en blanco o comentarios (usando el parser de tokens `untyped` el código fuente puede usar comentarios como en C++ o Java sin esfuerzo adicional.)

b) Se utilizan combinadores que trabajan sobre tokens. Por ejemplo, el parser `reservedOp lis "+"`, parsea el operador +, `reserved lis "if"` parsea la palabra reservada if, para parsear un identificador se puede utilizar `identifier lis`, y el parser `parens lis p` parsea lo mismo que `p`, pero entre paréntesis. Se recomienda no mezclar los operadores que trabajan a bajo nivel con los operadores que trabajan sobre tokens ya que pueden surgir problemas, por ejemplo, con el manejo de los espacios en blanco. Muchos combinadores son similares a los de la biblioteca simple vista en clase, por ejemplo `many`, `many1`, y $<|>$. El combinador $<|>$ es diferente en Parsec ya que, para mejorar la eficiencia, sólo va a tratar de ejecutar el segundo parser si el primero no consumió nada de la entrada. Por lo tanto, si dos opciones pueden comenzar con el mismo caracter, es conveniente usar el combinador `try`, donde `try p` se comporta como `p` excepto que si `p` falla no consume elementos de la entrada. Otro combinador útil que se recomienda utilizar es `chainl1`, que permite parsear operadores asociativos a izquierda, pero evitando la recursión a izquierda (ver su documentación en el enlace de más arriba). En `Main.hs`, se puede cambiar la última ĺnea para elegir entre imprimir el AST parseado (para probar el parser), e imprimir el resultado de la evaluación (cuando se implemente el evaluador).

2.4 Semántica Denotacional para Expresiones

Para definir la semántica de las expresiones enteras y booleanas de la gramática abstracta, se deben definir funciones semánticas que les asignen un significado. El significado de una expresión entera es un valor de $\mathbb{Z}$, y el significado de una expresión booleana es un valor en $\mathbb{B} = \{\mathbf{true}, \mathbf{false}\}$. El significado o denotación de cada expresión depende de los valores de sus variables, de un estado, que representamos como una función que le atribuye a cada variable un valor entero. Si llamamos Σ, al conjunto de estados $\langle var \rangle \to \mathbb{Z}$, las funciones semánticas para las expresiones del lenguaje son

$$[\![-]\!]_{\text{intexp}} \in \langle intexp \rangle \to \Sigma \to \mathbb{Z} \qquad [\![-]\!]_{\text{boolexp}} \in \langle boolexp \rangle \to \Sigma \to \mathbb{B}$$

y quedan definidas por las siguientes ecuaciones:

$$
\begin{array}{lcll}
[\![0]\!]_{\text{intexp}}\sigma & = & 0 & \text{(y análogamente para 1, 2, ...)} \\
[\![v]\!]_{\text{intexp}}\sigma & = & \sigma v & \\
[\![-_u e]\!]_{\text{intexp}}\sigma & = & -[\![e]\!]_{\text{intexp}}\sigma & \\
[\![e_0 + e_1]\!]_{\text{intexp}}\sigma & = & [\![e_0]\!]_{\text{intexp}}\sigma + [\![e_1]\!]_{\text{intexp}}\sigma & \text{(y análogamente para } -_b, \times \text{ y } \div) \\
 & & & \\
[\![\textbf{true}]\!]_{\text{boolexp}}\sigma & = & \textbf{true} & \\
[\![\textbf{false}]\!]_{\text{boolexp}}\sigma & = & \textbf{false} & \\
[\![e_0 = e_1]\!]_{\text{boolexp}}\sigma & = & [\![e_0]\!]_{\text{intexp}}\sigma = [\![e_1]\!]_{\text{intexp}}\sigma & \text{(y análogamente para } < \text{ y } >) \\
[\![\neg p]\!]_{\text{boolexp}}\sigma & = & \neg[\![p]\!]_{\text{boolexp}}\sigma & \\
[\![p_0 \wedge p_1]\!]_{\text{boolexp}}\sigma & = & [\![p_0]\!]_{\text{boolexp}}\sigma \wedge [\![p_1]\!]_{\text{boolexp}}\sigma & \\
[\![p_0 \vee p_1]\!]_{\text{boolexp}}\sigma & = & [\![p_0]\!]_{\text{boolexp}}\sigma \vee [\![p_1]\!]_{\text{boolexp}}\sigma &
\end{array}
$$

Es importante distiguir entre el lenguaje del cual se describe la semántica, o *lenguaje objeto*, y el lenguaje que se utiliza para describirla, el *metalenguaje*. En el lado izquierdo de cada ecuación, los corchetes dobles encierran un *patrón* similar al lado derecho de alguna regla de producción de la gramática abstracta, donde e, e_0 y e_1 son metavariables sobre expresiones enteras y p, p_0 y p_1 son metavariables sobre expresiones booleanas.

Puede pensarse que es absurdo definir 0 en términos de 0, + en términos de +, y así sucesivamente. Sin embargo, no hay circularidad en las definiciones porque los símbolos encerrados entre corchetes dobles denotan *constructores* del lenguaje objeto, mientras que fuera de ellos denotan operadores del metalenguaje (en este caso, la matemática y lógica convencionales).

Las ecuaciones dadas satisfacen dos condiciones fundamentales:

- Existe exactamente una ecuación para cada producción de la gramática abstracta
- Cada ecuación expresa el significado de una frase en función de los significados de sus subfrases

Un conjunto de ecuaciones que cumple estas condiciones se dice que es *dirigido por sintaxis*, y en conjunto con una definición adecuada de la gramática asegura que los objetos definidos son realmente funciones. Existe un solo problema con estas definiciones: cada expresión entera debe denotar algún valor entero, pero en la aritmética convencional no se le puede asignar ningún valor con sentido a una división de la forma $n \div 0$. Por simplicidad, evitamos tratar este problema dentro de la definición de la semántica denotacional, pero sin embargo lo trataremos más elegantemente al momento de construir un intérprete para el lenguaje.

Ejercicio 2.4.1 *Extienda la semántica denotacional para expresiones enteras para incluir el operador ternario descripto en el Ejercicio 2.2.1*

2.5 Semántica Operacional Estructural para Comandos

La ejecución de un comando puede modelarse mediante una secuencia

$$\gamma_0 \rightsquigarrow \gamma_1 \rightsquigarrow \cdots$$

donde cada *configuración* γ_i es

- un estado (una configuración terminal)

- un comando junto con un estado (una configuración no terminal)

La semántica operacional de LIS se describe en términos de:

$\Gamma_{\mathrm{N}} = \langle comm \rangle \times \Sigma$, el conjunto de configuraciones no terminales

$\Gamma_{\mathrm{T}} = \Sigma$, el conjunto de configuraciones terminales

$\Gamma = \Gamma_{\mathrm{N}} \cup \Gamma_{\mathrm{T}}$, el conjunto de todas las configuraciones

$\rightsquigarrow$, la relación de transicion de Γ_{N} a Γ

$\rightsquigarrow^*$, la clausura transitiva de $\rightsquigarrow$, donde $\gamma \rightsquigarrow^* \gamma'$ si existe una ejecución finita que comienza en γ y termina en γ'.

$[\, f \mid x{:}\, e \,]$, que denota la función f', tal que $\operatorname{dom} f' = \operatorname{dom} f \cup \{x\}$, $f'x = e$, y $\forall y \in \operatorname{dom} f \setminus \{x\} \,.\, f'y = f\ y$

Se utilizan reglas de inferencia para describir la relación de transición, utilizando la semántica denotacional de la sección anterior para las expresiones. Una ejecución $\gamma \rightsquigarrow \gamma$ es válida si y sólo si puede probarse como consecuencia de las siguientes reglas de inferencia,

$$\frac{}{\langle v := e, \sigma \rangle \rightsquigarrow [\, \sigma \mid v{:}\, [\![e]\!]_{\mathrm{intexp}} \sigma \,]}\ \mathrm{ASS}$$

$$\frac{}{\langle \mathbf{skip}, \sigma \rangle \rightsquigarrow \sigma}\ \mathrm{SKIP}$$

$$\frac{\langle c_0, \sigma \rangle \rightsquigarrow \sigma'}{\langle c_0; c_1, \sigma \rangle \rightsquigarrow \langle c_1, \sigma' \rangle}\ \mathrm{SEQ}_1 \qquad \frac{\langle c_0, \sigma \rangle \rightsquigarrow \langle c_0', \sigma' \rangle}{\langle c_0; c_1, \sigma \rangle \rightsquigarrow \langle c_0'; c_1, \sigma' \rangle}\ \mathrm{SEQ}_2$$

Cuando $[\![b]\!]_{\mathrm{boolexp}} \sigma = \mathbf{true}$:

$$\frac{}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle \rightsquigarrow \langle c_0, \sigma \rangle}\ \mathrm{IF}_1 \qquad \frac{}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma \rangle \rightsquigarrow \langle c; \mathbf{while}\ b\ \mathbf{do}\ c, \sigma \rangle}\ \mathrm{WHILE}_1$$

Cuando $[\![b]\!]_{\mathrm{boolexp}} \sigma = \mathbf{false}$:

$$\frac{}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle \rightsquigarrow \langle c_1, \sigma \rangle}\ \mathrm{IF}_2 \qquad \frac{}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma \rangle \rightsquigarrow \sigma}\ \mathrm{WHILE}_2$$

La relación $\rightsquigarrow^*$, la clausura transitiva de la relación $\rightsquigarrow$, también se puede definir usando reglas de inferencia:

$$\frac{\gamma \rightsquigarrow \gamma'}{\gamma \rightsquigarrow^* \gamma'}\ \mathrm{TR}_1 \qquad \frac{\gamma \rightsquigarrow^* \gamma' \quad \gamma' \rightsquigarrow^* \gamma''}{\gamma \rightsquigarrow^* \gamma''}\ \mathrm{TR}_2 \qquad \frac{}{\gamma \rightsquigarrow^* \gamma}\ \mathrm{TR}_3$$

Puede demostrarse que la relación de transición definida por las reglas de inferencia dadas es una función, es decir, que para cada configuración inicial γ existe exactamente una ejecución de longitud máxima que termina en una configuración terminal o es infinita (en cuyo caso diremos que γ diverge, y lo notamos con $\gamma \uparrow$). La semántica denotacional para los comandos de LIS está dada por la función:

$$[\![-]\!]_{\mathrm{comm}} \in \langle comm \rangle \to \Sigma \to \Sigma \cup \{\bot\}$$

$$[\![c]\!]_{\text{comm}}\sigma = \begin{cases} \bot & \text{si } \langle c, \sigma\rangle \uparrow \\ \sigma' & \text{si } \langle c, \sigma\rangle \rightsquigarrow^* \sigma' \end{cases}$$

El hecho de que la relación $\rightsquigarrow$ sea una función (una función total más específicamente), significa que a partir de una configuración se puede alcanzar en un solo paso exactamente una configuración. Esta propiedad, junto con el hecho de que las funciones semánticas para expresiones son dirigidas por sintaxis, permite construir fácilmente un intérprete de LIS en un lenguaje funcional con ajuste de patrones como Haskell.

Ejercicio 2.5.1 *Utilizando las reglas de inferencia, construya un árbol de prueba para*

$$\langle x := x + 1; \textbf{if } x > 0 \textbf{ then skip else } x := x - 1, [\,\sigma \mid x{:}\,0\,]\rangle \rightsquigarrow^* [\,\sigma \mid x{:}\,1\,]$$

Si utiliza LaTeX, puede utilizar el paquete `Proof` *para generar el árbol.*

2.6 Intérpretes monádicos

Ejercicio 2.6.1 *Complete el script bosquejado en el archivo* `Eval1.hs`*, para construir un intérprete de LIS dejando que el metalenguaje (*`Haskell`*) maneje los errores de división por 0 y de inexistencia de variables. Puede utilizar la función* `run` *definida en* `Main.hs` *para verificar que el intérprete se comporta como es esperado al ejecutar los programas de ejemplo sqrt.lis, error1.lis y error2.lis*

El evaluador simple se encuentra parcialmente implementado en `src/Eval1.hs`. El estado del programa se representa mediante el tipo de datos `Env`, que es simplemente una lista de pares de nombres de variable y sus respectivos valores. Se utiliza una mónada de estado, llamada `State`, para representar una computación que tiene acceso al estado del programa:

```
newtype State a = State {runState :: Env -> (a, Env)}
instance Monad State where
   return x = State (\s -> (x , s))
   m >>= f = State ( \s -> let (v,s0) = runState m s
                           in runState (f v) s0)
```

Una computación con estado es una función que recibe un estado original, computa algún valor y retorna un nuevo estado. Notar que en este caso no alcanza con la mónada `Reader`, ya que al ejecutar una asignación necesitamos modificar el entorno. La clase `MonadState` tiene las operaciones necesarias a implementar en mónadas con posibilidad de manejar variables con valores enteros.

```
class Monad m => MonadState m where
    lookfor :: Variable -> m Int
    update :: Variable -> Int -> m ()
```

Ejercicio 2.6.2 *Cree un archivo* `Eval2.hs` *y reimplemente el evaluador modificando el tipo de retorno y la definición de la función de evaluación para poder distinguir cuando se producen errores, mostrando un mensaje acorde al error producido. Modifique* `Main.hs` *para que importe* `Eval2` *en lugar de* `Eval1`

La mónada utilizada para representar computaciones con estado de variables y posibilidad de error será:

```
newtype StateError a = StateError {runStateError :: Env -> Maybe (a,Env)}
```

Con esta nueva definición podremos marcar errores devolviendo un `Nothing`. Agregamos además una clase `MonadError` para representar las operaciones de aquellas mónadas que pueden producir errores.

```
class Monad m => MonadError m where
  throw :: m a
```

Ejercicio 2.6.3 *Cree un archivo* `Eval3.hs` *y reimplemente el evaluador en* `Eval2.hs` *para que además de detectar errores, devuelva el resultado junto con la cantidad de operaciones* $+$, $-$, $*$ *y* $/$

Para esto, deberá proponer una modificación de la mónada `StateError` que lleve un entero donde se cuenten las operaciones aritméticas.

Ejercicio 2.6.4 *Escriba un programa* `euclides.lis` *que use el algoritmo de Euclides para calcular el máximo común divisor de dos naturales a y b dados y verifique que también se interpreta correctamente.*

Ejercicio 2.6.5 *El comando* **repeat** *tiene la forma* **repeat** *c* **until** *b. Su efecto se describe por el siguiente diagrama de flujo:*

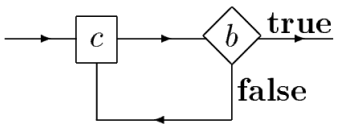

Figure 1: Comando **repeat**

Agregue una regla de producción a la gramática abstracta de LIS para el comando **repeat** *y de una descripción de la semántica del comando utilizando reglas de inferencia (hay más de una descripción posible).*