Trabajo práctico 1

1 Introducción

Se presenta un lenguaje imperativo simple con variables enteras y comandos para asignación, composición secuencial, ejecución condicional (**if**) y ciclos (**while**). Se especifica su sintaxis abstracta, su sintaxis concreta, una realización de su sintaxis abstracta en Haskell, y por último su semántica operacional big-step de expresiones y small-step de comandos. El objetivo del trabajo es construir un intérprete en Haskell para el lenguaje presentado. El trabajo se debe realizar en grupos de dos personas y la fecha límite de entrega es el Jueves 5 de septiembre, en donde se debe entregar:

- en papel, un informe con los ejercicios resueltos incluyendo **todo el código** que haya escrito;
- en forma electrónica (en un archivo comprimido) el código fuente de los intérpretes (archivos `Evali.hs`) y del parser (`Parser.hs`), usando el sitio de la materia del campus virtual de la UNR (`http://comunidades.campusvirtualunr.edu.ar`).

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:

$$
\begin{array}{rcl}
\mathit{intexp} & ::= & \mathit{nat} \mid \mathit{var} \mid -_u \ \mathit{intexp} \\
& \mid & \mathit{intexp} \ + \mathit{intexp} \\
& \mid & \mathit{intexp} \ -_b \ \mathit{intexp} \\
& \mid & \mathit{intexp} \ \times \ \mathit{intexp} \\
& \mid & \mathit{intexp} \ \div \ \mathit{intexp} \\
\mathit{boolexp} & ::= & \textsf{true} \mid \textsf{false} \\
& \mid & \mathit{intexp} \ == \ \mathit{intexp} \\
& \mid & \mathit{intexp} \ \neq \ \mathit{intexp} \\
& \mid & \mathit{intexp} \ < \ \mathit{intexp} \\
& \mid & \mathit{intexp} \ > \ \mathit{intexp} \\
& \mid & \mathit{boolexp} \wedge \mathit{boolexp} \\
& \mid & \mathit{boolexp} \vee \mathit{boolexp} \\
& \mid & \neg \ \mathit{boolexp} \\
\mathit{comm} & ::= & \textsf{skip} \\
& \mid & \mathit{var} = \mathit{intexp} \\
& \mid & \mathit{comm}; \mathit{comm} \\
& \mid & \textsf{if} \ \mathit{boolexp} \ \textsf{then} \ \mathit{comm} \\
& \mid & \textsf{if} \ \mathit{boolexp} \ \textsf{then} \ \mathit{comm} \ \textsf{else} \ \mathit{comm} \\
& \mid & \textsf{while} \ \mathit{boolexp} \ \textsf{do} \ \mathit{comm}
\end{array}
$$

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*
| *intexp* `'>'` *intexp*
| *boolexp* `'&&'` *boolexp*
| *boolexp* `'||'` *boolexp*
| `'!'` *boolexp*
| `'('` *boolexp* `')'`
comm ::= **skip**
| *var* `'='` *intexp*
| *comm* `';'` *comm*
| `'if'` *boolexp* `'{'` *comm* `'}'`
| `'if'` *boolexp* `'{'` *comm* `'}'` `'else'` `'{'` *comm* `'}'`
| `'while'` *boolexp* `'{'` *comm* `'}'`

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

$$-_u \quad (* \ /) \quad (+ \ -_b) \quad (== \ != \ < \ >) \quad ! \quad \&\& \quad || \quad = \quad ;$$

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 1. Extienda las sintaxis abstracta y concreta de LIS para incluir asignaciones de variables como expresiones enteras, de la forma $x = e$ y el operador **,** para escribir una secuencia de expresiones enteras. Estas dos extensiones permitirán tener en el lenguaje secuencias de expresiones al estilo del lenguaje C, como por ejemplo $x = 3, y = x + 1, 7 * 2$, donde el valor de toda la expresión es el valor de la última expresión de la secuencia.

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

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

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

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

Los identificadores de variables podemos representarlos como Strings.

```
type Variable = String
```

Las expresiones aritméticas con el tipo IntExp y las booleanas con el tipo BoolExp

```
data IntExp = Const Integer
            | Var Variable
            | UMinus IntExp
            | Plus IntExp IntExp
            | Minus IntExp IntExp
            | Times IntExp IntExp
            | Div IntExp IntExp
data BoolExp = BTrue
             | BFalse
             | Eq IntExp IntExp
             | NEq IntExp IntExp
             | Lt IntExp IntExp
             | Gt IntExp IntExp
             | And BoolExp BoolExp
             | Or BoolExp BoolExp
             | Not BoolExp
```

Los comandos son representados por el tipo Comm. Notar que sólo se permiten variables de un tipo (entero).

```
data Comm = Skip
          | Let Variable IntExp
          | Seq Comm Comm
          | IfThenElse BoolExp Comm Comm
          | While BoolExp Comm
```

Ejercicio 2. Extienda la realización de la sintaxis abstracta en Haskell para incluir la asignación como expresiones y el operador **,** para secuencia de expresiones.

Ejercicio 3. 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 línea, 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 $\langle|\rangle$. El combinador $\langle|\rangle$ 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 línea 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 Operacional Big-Step para Expresiones

Para definir la semántica de las expresiones enteras y booleanas de la gramática abstracta, utilizaremos una semántica operacional de paso grande. Los valores de las expresiones enteras se definen de la siguiente manera:

$$nv ::= 0 \mid np \mid -_u\ np$$

donde np son los naturales positivos. Los valores booleanos por otra parte son

$$bv ::= \textbf{true} \mid \textbf{false}$$

El significado de cada expresión depende de un estado que le asigna un valor (entero) a sus variables. Llamamos Σ al conjunto de estados que le atribuye a cada variable un valor entero.
Definimos las relaciones de evaluación para las expresiones enteras y booleanas inductivamente mediante las siguientes reglas, donde $\sigma \in \Sigma$.

$$\frac{}{\langle nv, \sigma\rangle \Downarrow_{\text{intexp}} nv}\ \textsc{NVal} \qquad \frac{}{\langle v, \sigma\rangle \Downarrow_{\text{intexp}} \sigma v}\ \textsc{Var} \qquad \frac{\langle e, \sigma\rangle \Downarrow_{\text{intexp}} n}{\langle -_u e, \sigma\rangle \Downarrow_{\text{intexp}} -n}\ \textsc{UMinus}$$

$$\frac{\langle e_0, \sigma\rangle \Downarrow_{\text{intexp}} n_0 \quad \langle e_1, \sigma\rangle \Downarrow_{\text{intexp}} n_1}{\langle e_0 + e_1, \sigma\rangle \Downarrow_{\text{intexp}} n_0 + n_1}\ \textsc{Plus} \qquad \text{(análogamente para } -_b,\ \times\text{)}$$

$$\frac{\langle e_0, \sigma\rangle \Downarrow_{\text{intexp}} n_0 \quad \langle e_1, \sigma\rangle \Downarrow_{\text{intexp}} n_1 \quad n_1 \neq 0}{\langle e_0 \div e_1, \sigma\rangle \Downarrow_{\text{intexp}} n_0 \div n_1}\ \textsc{Div}$$

$$\frac{\langle e_0, \sigma\rangle \Downarrow_{\text{intexp}} n_0 \quad \langle e_1, \sigma\rangle \Downarrow_{\text{intexp}} n_1}{\langle e_0 == e_1, \sigma\rangle \Downarrow_{\text{boolexp}} n_0 = n_1}\ \textsc{Eq} \qquad \text{(análogamente para } <\ ,\ >\ \text{y}\ \neq\text{)}$$

$$\frac{}{\langle bv, \sigma\rangle \Downarrow_{\text{boolexp}} bv}\ \textsc{BVal} \qquad \frac{\langle p, \sigma\rangle \Downarrow_{\text{boolexp}} b}{\langle \neg p, \sigma\rangle \Downarrow_{\text{boolexp}} \neg b}\ \textsc{Not}$$

$$\frac{\langle p_0, \sigma\rangle \Downarrow_{\text{boolexp}} b_0 \quad \langle p_1, \sigma\rangle \Downarrow_{\text{boolexp}} b_1}{\langle p_0 \vee p_1, \sigma\rangle \Downarrow_{\text{boolexp}} b_0 \vee b_1}\ \textsc{Or} \qquad \text{(análogamente para } \wedge\text{)}$$

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 la relación semántica, el primer elemento del par encierra 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.
No hay circularidad en las definiciones porque los operadores del lado izquierdo de la relación semántica (en el dominio) denotan *constructores* del lenguaje objeto, mientras que del lado derecho (recorrido de la relación) denotan operadores del metalenguaje sobre *valores*.

Ejercicio 4. Extienda la semántica big-step de expresiones enteras para incluir la asignación de expresiones como expresiones y el operador **,** para secuencias de expresiones descriptos en el Ejercicio 2.2.
Dado que las expresiones podrán modificar el entorno de variables, utilizar la notación $[\, f \mid x : e \,]$ para denotar la función f', tal que $\text{dom}\, f' = \text{dom}\, f \cup \{x\}$, $f' x = e$, y $\forall y \in \text{dom}\, f \setminus \{x\}\ .\ f' y = f\ y$.

2.5 Semántica Operacional Estructural para Comandos

La semántica operacional de LIS se describe en términos de:

$\Gamma_{\mathrm{N}} = comm \times \Sigma$, 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 transición 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'.

Se utilizan reglas de inferencia para describir la relación de transición, utilizando la semántica operacional 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,

$$\frac{\langle e, \sigma\rangle \Downarrow_{\mathrm{intexp}} n}{\langle v := e, \sigma\rangle \rightsquigarrow [\,\sigma \mid v\colon n\,]}\ \textsc{Ass} \qquad \frac{}{\langle \textbf{skip}, \sigma\rangle \rightsquigarrow \sigma}\ \textsc{Skip}$$

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

$$\frac{\langle b, \sigma\rangle \Downarrow_{\mathrm{boolexp}} \textbf{true}}{\langle \textbf{if } b \textbf{ then } c_0, \sigma\rangle \rightsquigarrow \langle c_0, \sigma\rangle}\ \textsc{If}_1 \qquad \frac{\langle b, \sigma\rangle \Downarrow_{\mathrm{boolexp}} \textbf{false}}{\langle \textbf{if } b \textbf{ then } c_0, \sigma\rangle \rightsquigarrow \sigma}\ \textsc{If}_2$$

$$\frac{\langle b, \sigma\rangle \Downarrow_{\mathrm{boolexp}} \textbf{true}}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \rightsquigarrow \langle c_0, \sigma\rangle}\ \textsc{If}_3 \qquad \frac{\langle b, \sigma\rangle \Downarrow_{\mathrm{boolexp}} \textbf{false}}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \rightsquigarrow \langle c_1, \sigma\rangle}\ \textsc{If}_4$$

$$\frac{\langle b, \sigma\rangle \Downarrow_{\mathrm{boolexp}} \textbf{true}}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \rightsquigarrow \langle c; \textbf{while } b \textbf{ do } c, \sigma\rangle}\ \textsc{While}_1 \qquad \frac{\langle b, \sigma\rangle \Downarrow_{\mathrm{boolexp}} \textbf{false}}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \rightsquigarrow \sigma}\ \textsc{While}_2$$

Ejercicio 5. ¿Es la relación de evaluación de un paso $\rightsquigarrow$ determinista? Si lo es, demostrarlo. Caso contrario, proveer un contraejemplo.

Ejercicio 6. Utilizando las reglas de inferencia, construya un árbol de derivación para probar que el siguiente juicio es válido.

$$\langle x = x - 1; \textbf{while } x > 0 \textbf{ do } x = x - 2, [\,\sigma \mid x\colon 2\,]\rangle \rightsquigarrow^* [\,\sigma \mid x\colon -1\,]$$

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

Ejercicio 7. 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`

Ejercicio 8. 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. Por ejemplo, podemos considerar errores de división por 0 y de indefinición de variables mediante un tipo Error

```
data Error = DivByZero | UndefVar
```

y hacer que el evaluador devuelva un tipo Either Err a, donde a es el tipo de retorno de la función. Modifique `Main.hs` para que importe `Eval2` en lugar de `Eval1`.

Ejercicio 9. Cree un archivo `Eval3.hs` y reimplemente el evaluador en `Eval2.hs` para que además de detectar errores, devuelva el resultado junto con el costo de las operaciones realizadas. El costo de cada operación está dado en la siguiente tabla:

$$
\begin{array}{rcll}
\mathrm{W}(e_1\ nop\ e_2) & = & 2 + \mathrm{W}(e_1) + \mathrm{W}(e_2) & \text{donde } nop \in \{\div, \times\} \\
\mathrm{W}(e_1\ bop\ e_2) & = & 1 + \mathrm{W}(e_1) + \mathrm{W}(e_2) & \text{donde } bop \in \{+, -_b, \wedge, \vee, >, <, ==, \neq\} \\
\mathrm{W}(op\ e) & = & 1 + \mathrm{W}(e) & \text{donde } op \in \{-_u, \neg\}
\end{array}
$$

Ejercicio 10. El comando **for** es usado al igual que **while** para la repetición un código. La ejecución de **for** $(e_1\ ;\ e_2\ ;\ e_3)\ c$ produce el siguiente efecto: ejecuta e_1 sólo una vez antes de entrar al ciclo, mientras la expresión e_2 sea cierta ejecuta el código c y luego e_3. El ciclo termina cuando e_2 es falsa.
Agregue una regla de producción a la gramática abstracta de LIS y extienda la semántica operacional de comandos para el comando **for**.