Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Escuela de Ciencias Exactas y Naturales
Departamento de Ciencias de la Computación
Análisis de Lenguajes de Programación

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 denotacional de expresiones y semántica operacional de comandos. El objetivo del trabajo es construir un intérprete en Haskell para el lenguaje presentado. Para simplificar el trabajo, se brinda un analizador sintáctico –*parser*– que traduce un programa de su representación concreta a una representación no ambigua en términos de la sintaxis abstracta.

El trabajo se puede realizar en grupos de hasta dos personas y siendo la fecha límite de entrega el Jueves 3 de septiembre:

- en papel, un informe con los ejercicios resueltos incluyendo **todo el código** que haya escrito;
- por correo electrónico a `entregas.alp@gmail.com` con asunto `TP1 ALP`, el código fuente del intérprete (archivos `Eval`i`.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:

$$
\begin{aligned}
\langle intexp\rangle ::= {} & \langle nat\rangle \mid \langle var\rangle \mid -_u\langle intexp\rangle \\
\mid {} & \langle intexp\rangle + \langle intexp\rangle \\
\mid {} & \langle intexp\rangle -_b \langle intexp\rangle \\
\mid {} & \langle intexp\rangle \times \langle intexp\rangle \\
\mid {} & \langle intexp\rangle \div \langle intexp\rangle \\
\langle boolexp\rangle ::= {} & \mathbf{true} \mid \mathbf{false} \\
\mid {} & \langle intexp\rangle = \langle intexp\rangle \\
\mid {} & \langle intexp\rangle < \langle intexp\rangle \\
\mid {} & \langle intexp\rangle > \langle intexp\rangle \\
\mid {} & \langle boolexp\rangle \wedge \langle boolexp\rangle \\
\mid {} & \langle boolexp\rangle \vee \langle boolexp\rangle \\
\mid {} & \neg\langle boolexp\rangle \\
\langle comm\rangle ::= {} & \mathbf{skip}
\end{aligned}
$$

| $\langle var \rangle := \langle intexp \rangle$
| $\langle comm \rangle ; \langle comm \rangle$
| **if** $\langle boolexp \rangle$ **then** $\langle comm \rangle$ **else** $\langle comm \rangle$
| **while** $\langle boolexp \rangle$ **do** $\langle comm \rangle$

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:

$\langle digit \rangle$ ::= `'0'` | `'1'` | ... | `'9'`
$\langle letter \rangle$ ::= `'a'` | ... | `'Z'`
$\langle nat \rangle$ ::= $\langle digit \rangle$ | $\langle digit \rangle\langle nat \rangle$
$\langle var \rangle$ ::= $\langle letter \rangle$ | $\langle letter \rangle\langle var \rangle$
$\langle intexp \rangle$ ::= $\langle nat \rangle$
| $\langle var \rangle$
| `'-'`$\langle intexp \rangle$
| $\langle intexp \rangle$`'+'`$\langle intexp \rangle$
| $\langle intexp \rangle$`'-'`$\langle intexp \rangle$
| $\langle intexp \rangle$`'*'`$\langle intexp \rangle$
| $\langle intexp \rangle$`'/'`$\langle intexp \rangle$
| `'('`$\langle intexp \rangle$`')'`
$\langle boolexp \rangle$ ::= `'true'` | `'false'`
| $\langle intexp \rangle$`'='`$\langle intexp \rangle$
| $\langle intexp \rangle$`'<'`$\langle intexp \rangle$
| $\langle intexp \rangle$`'>'`$\langle intexp \rangle$
| $\langle boolexp \rangle$`'&'`$\langle boolexp \rangle$
| $\langle boolexp \rangle$`'|'`$\langle boolexp \rangle$
| `'~'`$\langle boolexp \rangle$
| `'('`$\langle boolexp \rangle$`')'`
$\langle comm \rangle$ ::= `'skip'`
| $\langle var \rangle$`':='`$\langle intexp \rangle$
| $\langle comm \rangle$`';'`$\langle comm \rangle$
| `'if'` $\langle boolexp \rangle$ `'then'` $\langle comm \rangle$ `'else'` $\langle comm \rangle$ `'end'`
| `'while'` $\langle boolexp \rangle$ `'do'` $\langle comm \rangle$ `'end'`

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:

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

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 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

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

Los identificadores de variables podemos representarlos como *String*s.

```
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
             | 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
          | 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

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 un estado que le asigna un valor (entero)

a sus variables. Llamamos Σ al conjunto de estados $var \to \mathbb{Z}$ que le atribuye a cada variable un valor entero. Las funciones semánticas para las expresiones del lenguaje son

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

y quedan definidas por las siguientes ecuaciones, donde $\sigma \in \Sigma$.

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

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:

$\Gamma_{\mathrm{N}} = comm \times \Sigma$, 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 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 $\mathrm{dom}\, f' = \mathrm{dom}\, f \cup \{x\}$, $f'x = e$, y $\forall y \in \mathrm{dom}\, f \setminus \{x\} \,.\, f'y = f\ y$

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,

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

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

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

$$\frac{[\![b]\!]_{\mathrm{boolexp}}\sigma = \mathbf{true}}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma\rangle \rightsquigarrow \langle c_0, \sigma\rangle}\ \mathrm{IF}_1 \qquad \frac{[\![b]\!]_{\mathrm{boolexp}}\sigma = \mathbf{true}}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma\rangle \rightsquigarrow \langle c; \mathbf{while}\ b\ \mathbf{do}\ c, \sigma\rangle}\ \mathrm{WHILE}_1$$

$$\frac{[\![b]\!]_{\mathrm{boolexp}}\sigma = \mathbf{false}}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma\rangle \rightsquigarrow \langle c_1, \sigma\rangle}\ \mathrm{IF}_2 \qquad \frac{[\![b]\!]_{\mathrm{boolexp}}\sigma = \mathbf{false}}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma\rangle \rightsquigarrow \sigma}\ \mathrm{WHILE}_2$$

Ejercicio 2.5.1. Demostrar que la relación de evaluación de un paso $\rightsquigarrow$ es determinista.

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

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

Como la relación de evaluación $\rightsquigarrow$ es determinista, $\rightsquigarrow^*$ también lo es. Es decir, 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:

$$[\![-]\!]_{\text{comm}} \in comm \to \Sigma \to \Sigma \cup \{\bot\}$$

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

El símbolo $\bot$ expresa un valor del dominio de interpretación que presenta a las computaciones divergentes. Que la relación $\rightsquigarrow$ sea determinista, 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 2.5.2. 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.

Ejercicio 2.5.3. 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.

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` y `error.lis`

Ejercicio 2.5.4. 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 produce un error de división por 0 (modifique `Main.hs` para que importe `Eval2` en lugar de `Eval1`.)

Ejercicio 2.5.5. 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 aritméticas ($+$, $-$, $*$, y $/$) realizadas durante la evaluación.

Ejercicio 2.5.6. El comando **repeat** tiene la forma **repeat** c **until** b. Su efecto se describe por el siguiente diagrama de flujo:

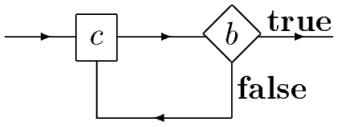

Figure 1: Comando **repeat**

Agregue una regla de producción a la gramática abstracta de LIS y extienda la semántica operacional de comandos para el comando **repeat**.