



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 19 de septiembre, en donde se debe entregar:

- un informe en PDF con los ejercicios resueltos incluyendo **todo el código** que haya escrito;
- un archivo comprimido del directorio **src**, usando el sitio de la materia del campus virtual de la UNR (<http://comunidades.campusvirtualunr.edu.ar>).

Como complemento a esta consigna, el trabajo práctico incluye un **README** con aclaraciones técnicas sobre la implementación propuesta. Se recomienda leer este documento con atención y consultar a algún docente cualquier duda.

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:

```
intexp ::= nat | var |  $-_u$  intexp
        | intexp + intexp
        | intexp  $-_b$  intexp
        | intexp  $\times$  intexp
        | intexp  $\div$  intexp

boolexp ::= true | false
        | intexp == intexp
        | intexp  $\neq$  intexp
        | intexp < intexp
        | intexp > intexp
        | boolexp  $\wedge$  boolexp
        | boolexp  $\vee$  boolexp
        |  $\neg$  boolexp

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 programa 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 '}'

```

Notar que la construcción `'if' boolexp '{' comm '}'` de la sintaxis concreta no cuenta con una representación directa en la sintaxis abstracta. Esto se debe a que esta puede representarse de forma abstracta mediante el término **if boolexp then comm else skip**.

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 una nueva expresión entera, al estilo del operador condicional ternario `"?:"` del lenguaje C. Suponer que el nuevo operador tiene la menor precedencia respecto a los demás operadores.

2.3 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

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

Los identificadores de variables podemos representarlos como `Strings`.

```
type Variable = String
```

Las expresiones serán representadas con el tipo parametrizado `Exp a`, donde `Exp Int` corresponde a las expresiones aritméticas y `Exp Bool` las expresiones booleanas. Para una explicación más detallada de esta definición, ver el README.

```
data Exp a where
  Const  :: Int → Exp Int
  Var    :: Variable → Exp Int
  UMinus :: Exp Int → Exp Int
  Plus   :: Exp Int → Exp Int → Exp Int
  Minus  :: Exp Int → Exp Int → Exp Int
  Times  :: Exp Int → Exp Int → Exp Int
  Div    :: Exp Int → Exp Int → Exp Int
  BTrue  :: Exp Bool
  BFalse :: Exp Bool
  Lt     :: Exp Int → Exp Int → Exp Bool
  Gt     :: Exp Int → Exp Int → Exp Bool
  And    :: Exp Bool → Exp Bool → Exp Bool
  Or     :: Exp Bool → Exp Bool → Exp Bool
  Not    :: Exp Bool → Exp Bool
  Eq     :: Exp Int → Exp Int → Exp Bool
  NEq    :: Exp Int → Exp Int → Exp Bool
```

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

```
data Comm
  = Skip
  | Let Variable (Exp Int)
  | Seq Comm Comm
  | IfThenElse (Exp Bool) Comm Comm
  | While (Exp Bool) Comm
```

Ejercicio 2. Extienda la realización de la sintaxis abstracta en Haskell para incluir el operador ternario descrito en el Ejercicio 1. El nombre del constructor **debe** ser **ECond**.

En adelante realizar los ejercicios pedidos sobre el lenguaje con éstas extensiones.

Ejercicio 3. Implementar un parser en el archivo `src/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 carácter, `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 `<|>`. 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 carácter, 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).

El ejecutable implementado en `app/Main.hs` tiene opciones para imprimir el programa parseado, su AST, o el resultado de su evaluación (cuando los respectivos módulos sean implementados). Las primeras dos utilidades pueden ser de utilidad para probar el parser. Para más detalles sobre su funcionamiento, ver el `README`.

2.4 Semántica 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:

$$nv ::= int$$

es decir, son los números enteros. Los valores booleanos por otra parte son

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

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 la relación de evaluación para las expresiones inductivamente mediante las siguientes reglas, donde $\sigma \in \Sigma$.

$$\begin{array}{c} \frac{}{\langle nv, \sigma \rangle \Downarrow_{\text{exp}} nv} \text{ NVAL} \quad \frac{}{\langle x, \sigma \rangle \Downarrow_{\text{exp}} \sigma x} \text{ VAR} \quad \frac{\langle e, \sigma \rangle \Downarrow_{\text{exp}} n}{\langle -_u e, \sigma \rangle \Downarrow_{\text{exp}} -n} \text{ UMINUS} \\ \\ \frac{\langle e_0, \sigma \rangle \Downarrow_{\text{exp}} n_0 \quad \langle e_1, \sigma \rangle \Downarrow_{\text{exp}} n_1}{\langle e_0 + e_1, \sigma \rangle \Downarrow_{\text{exp}} n_0 + n_1} \text{ PLUS} \quad (\text{análogamente para } -, \times) \\ \\ \frac{\langle e_0, \sigma \rangle \Downarrow_{\text{exp}} n_0 \quad \langle e_1, \sigma \rangle \Downarrow_{\text{exp}} n_1 \quad n_1 \neq 0}{\langle e_0 \div e_1, \sigma \rangle \Downarrow_{\text{exp}} n_0 \div n_1} \text{ DIV} \\ \\ \frac{\langle e_0, \sigma \rangle \Downarrow_{\text{exp}} n_0 \quad \langle e_1, \sigma \rangle \Downarrow_{\text{exp}} n_1}{\langle e_0 == e_1, \sigma \rangle \Downarrow_{\text{exp}} n_0 = n_1} \text{ EQ} \quad (\text{análogamente para } <, > \text{ y } \neq) \\ \\ \frac{}{\langle bv, \sigma \rangle \Downarrow_{\text{exp}} bv} \text{ BVAL} \quad \frac{\langle p, \sigma \rangle \Downarrow_{\text{exp}} b}{\langle \neg p, \sigma \rangle \Downarrow_{\text{exp}} \neg b} \text{ NOT} \\ \\ \frac{\langle p_0, \sigma \rangle \Downarrow_{\text{exp}} b_0 \quad \langle p_1, \sigma \rangle \Downarrow_{\text{exp}} b_1}{\langle p_0 \vee p_1, \sigma \rangle \Downarrow_{\text{exp}} b_0 \vee b_1} \text{ OR} \quad (\text{análogamente para } \wedge) \end{array}$$

Es importante distinguir entre el lenguaje del cual se describe la semántica, o *lenguaje objeto*, y el lenguaje que se utiliza para describirla, el *metalinguaje*. 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*. Para no confundirlos, se escriben los segundos en negrita. Por ejemplo, en $\langle e_0 + e_1, \sigma \rangle \Downarrow_{\text{intexp}} n_0 + n_1$ El símbolo $+$ de la izquierda corresponde al terminal de la sintaxis abstracta de expresiones, mientras que $+$ es la suma de enteros.

Ejercicio 4. Extienda la semántica big-step de expresiones enteras para incluir el operador ternario descrito en el Ejercicio 1.

2.5 Semántica para Comandos

La semántica operacional de LIS se describe en términos de:

$\Gamma = \text{comm} \times \Sigma$, el conjunto de todas las configuraciones,

\rightsquigarrow , la relación de transición de Γ 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' .

Notar que toda ejecución que termina lo hace en $\langle \text{skip}, \sigma \rangle$ para algún estado σ .

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,

$$\begin{array}{c}
\frac{\langle e, \sigma \rangle \Downarrow_{\text{exp}} n}{\langle v = e, \sigma \rangle \rightsquigarrow \langle \text{skip}, [\sigma' \mid v: n] \rangle} \text{ASS} \\
\\
\frac{}{\langle \text{skip}; c_1, \sigma \rangle \rightsquigarrow \langle c_1, \sigma \rangle} \text{SEQ}_1 \quad \frac{\langle c_0, \sigma \rangle \rightsquigarrow \langle c'_0, \sigma' \rangle}{\langle c_0; c_1, \sigma \rangle \rightsquigarrow \langle c'_0; c_1, \sigma' \rangle} \text{SEQ}_2 \\
\\
\frac{\langle b, \sigma \rangle \Downarrow_{\text{exp}} \text{true}}{\langle \text{if } b \text{ then } c_0 \text{ else } c_1, \sigma \rangle \rightsquigarrow \langle c_0, \sigma \rangle} \text{IF}_1 \quad \frac{\langle b, \sigma \rangle \Downarrow_{\text{exp}} \text{false}}{\langle \text{if } b \text{ then } c_0 \text{ else } c_1, \sigma \rangle \rightsquigarrow \langle c_1, \sigma \rangle} \text{IF}_2 \\
\\
\frac{\langle b, \sigma \rangle \Downarrow_{\text{exp}} \text{true}}{\langle \text{while } b \text{ do } c, \sigma \rangle \rightsquigarrow \langle c; \text{while } b \text{ do } c, \sigma \rangle} \text{WHILE}_1 \quad \frac{\langle b, \sigma \rangle \Downarrow_{\text{exp}} \text{false}}{\langle \text{while } b \text{ do } c, \sigma \rangle \rightsquigarrow \langle \text{skip}, \sigma \rangle} \text{WHILE}_2
\end{array}$$

Ejercicio 5. ¿Es la relación de evaluación de un paso \rightsquigarrow determinista? Si lo es, demostrarlo (puede suponer que la relación \Downarrow es determinista). 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.

$$\langle x := x > y ? 2 * y : y; \text{while } x > 0 \text{ do } x := x - y, [[\sigma \mid x: 1] \mid y: 2] \rangle \rightsquigarrow^* \langle \text{skip}, [[\sigma \mid x: 0] \mid y: 2] \rangle$$

Si utiliza L^AT_EX, 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 opción `-e 1` del ejecutable 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. Reimplemente el evaluador en el archivo `Eval2.hs` 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 Error** a , donde a es el tipo de retorno de la función. Para correr este evaluador, debe usar la opción $-e$ 2.

Ejercicio 9. Reimplemente en el archivo `Eval3.hs` el evaluador en 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:

$$\begin{array}{lll}
 W(e_1 \text{ nop } e_2) & = & 3 + W(e_1) + W(e_2) & \text{donde } \text{nop} \in \{\div, \times\} \\
 W(e_1 \text{ bop } e_2) & = & 2 + W(e_1) + W(e_2) & \text{donde } \text{bop} \in \{+, -, \wedge, \vee, >, <, ==, \neq\} \\
 W(op \ e) & = & 1 + W(e) & \text{donde } op \in \{-u, \neg\}
 \end{array}$$

Ejercicio 10. El comando **repeat** tiene la forma **repeat** c **until** b . Su efecto se describe por el siguiente diagrama de flujo:

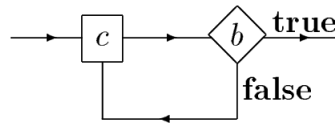


Figura 1: Comando **repeat**

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