Laboratorio de Programación Funcional 2018
Compilador e Intérprete de MicroPascal

La tarea consiste en la implementación de un compilador y un intérprete de una versión reducida y simplificada de Pascal, que llamaremos MicroPascal. El compilador deberá realizar las tareas de chequeo de tipos, optimización y generación de código de máquina equivalente al programa MicroPascal compilado. El intérprete debe poder interpretar las instrucciones del código de máquina y ejecutar las acciones correspondientes.

1 MicroPascal

La siguiente EBNF describe la sintaxis de MicroPascal, donde ⟨*ident*⟩ es un identificador válido y ⟨*nat*⟩ un número natural:

⟨*program*⟩ ::= '`program`' ⟨*ident*⟩ '`;`' ⟨*defs*⟩ ⟨*body*⟩ '`.`'

⟨*defs*⟩ ::= '`var`' {⟨*ident*⟩ '`:`' ⟨*type*⟩ '`;`'}

⟨*type*⟩ ::= '`integer`' | '`boolean`'

⟨*body*⟩ ::= '`begin`' [⟨*stmt*⟩{'`;`'⟨*stmt*⟩}] '`end`'

⟨*stmt*⟩ ::= ⟨*ident*⟩ '`:=`' ⟨*expr*⟩
| '`if`' ⟨*expr*⟩ '`then`' ⟨*body*⟩ '`else`' ⟨*body*⟩
| '`while`' ⟨*expr*⟩ '`do`' ⟨*body*⟩
| '`writeln`' '`(`' ⟨*expr*⟩ '`)`'
| '`readln`' '`(`' ⟨*ident*⟩ '`)`'

⟨*expr*⟩ ::= ⟨*ident*⟩ | '`true` | `false`' | ⟨*nat*⟩
| ⟨*expr*⟩ ⟨*bop*⟩ ⟨*expr*⟩ | ⟨*uop*⟩ ⟨*expr*⟩

⟨*bop*⟩ ::= '`or`' | '`and`' | '`=`' | '`<`' | '`+`' | '`-`' | '`*`' | '`div`' | '`mod`'

⟨*uop*⟩ ::= '`not`' | '`-`'

Un programa tiene un nombre, una lista (posiblemente vacía) de declaraciones de variables y un bloque **begin−end** compuesto de una secuencia de

```
program ejemplo1;
var
begin
end.
```

Figure 1: Programa más simple posible

```
program ejemplo2;
var x : integer;
    y : integer;
    b : boolean;
begin
   x := 10;
   y := x * (3 + 2);
   b := true;
   b := not (x < 10)
end.
```

Figure 2: Declaración de variables y asignación

instrucciones. En la Figura 1 se muestra el ejemplo de programa más simple que cumple con la sintaxis dada por la gramática de MicroPascal. Notar que, aunque no se declaren variables, la palabra clave **var** es obligatoria.

Al declarar una variable se especifica su nombre y su tipo. En MicroPascal hay dos tipos: **integer** y **boolean**. En la Figura 2 se puede ver un programa en el que se declaran variables de distintos tipos y luego se realizan asignaciones a dichas variables.

Las expresiones pueden ser variables (ej. x), literales booleanos, literales enteros, aplicación de un operador unario a una sub-expresión y la aplicación de un operador binario a dos sub-expresiones. Los operadores unarios son la negación $-$ y el no lógico **not**. Los operadores binarios son los operadores lógicos **or** y **and**, los operadores de comparación $=$ y $<$, y los operadores enteros $+$, $-$, $*$, **div** y **mod**.

Además de las asignaciones se tienen instrucciones de: selección **if**$-$**then**$-$**else** (no se puede omitir el **else**), iteración condicional **while**, y de entrada/salida **writeln** y **readln**. Los procedimientos de entrada/salida pueden tomar sólo un parámetro de tipo entero. En las figuras 3 y 4 se muestran ejemplos de programas que utilizan estas instrucciones.

El módulo `Syntax`, provisto en el archivo `Syntax.hs`, contiene un tipo algebraico de datos `Program` que representa a los árboles de sintaxis abstracta de programas MicroPascal y una función de parsing, que dada una cadena de caracteres con un programa MicroPascal retorna su árbol de sintaxis abstracta o los errores de sintaxis encontrados al intentar parsear:

```
parser :: String -> Either String Program
```

```
program ejemplo3;
var w : integer;
    x : boolean;
begin
   readln(w);
   x := w < 10;
   if x then
   begin
     writeln(w)
   end
   else
   begin
     writeln(10)
   end
end.
```

Figure 3: Instrucciones (If)

1.1 Chequeos

El compilador debe realizar chequeos de nombres y de tipos, como se describe a continuación. Estos chequeos deben ser realizados por la función del módulo `TypeChecker` que debe ser implementada como parte de la tarea:

```
checkProgram ::  Program -> [Error]
```

La función toma como entrada el árbol de sintaxis abstracta de un programa y retorna la lista de errores encontrados. Si el programa es correcto la lista es vacía.

1.1.1 Chequeo de Nombres

La primera etapa de chequeos consiste en el chequeo de nombres. Se debe verificar que no se usen nombres que no se hayan declarado y que no hayan múltiples declaraciones de un mismo nombre.

Por ejemplo, en el programa de la Figura 5 se detectan los siguientes errores de declaraciones duplicadas:

```
Duplicated definition: y
Duplicated definition: y
Duplicated definition: x
Duplicated definition: y
```

Notar que las repeticiones se listan a medida en que se van encontrando en una recorrida de arriba hacia abajo.

En el programa de la Figura 6 se detectan los usos de variables no definidas:

```
program ejemplo4;
var x : integer;
begin
   readln(x);
   while 0 < x do
   begin
     writeln(x);
     x := x - 1
   end
end.
```

Figure 4: Instrucciones (While)

```
program ejemplo5;
var x : integer;
    y : boolean;
    y : integer;
    y : integer;
    x : integer;
    y : boolean;
begin
   readln(y);
   x := x + y
end.
```

Figure 5: Definiciones Duplicadas

```
program ejemplo6;
var x : integer;
    y : integer;
begin
   readln(z);
   i := 1;
   while i < 10 do
   begin
      i := i + 1
   end
end.
```

Figure 6: Nombres no definidos

```
Undefined: z
Undefined: i
Undefined: i
Undefined: i
Undefined: i
```

Notar que se reporta cada uso del nombre no definido.

1.1.2 Chequeo de Tipos

En caso de no existir errores en la etapa de chequeo de nombres, se procede con el chequeo de tipos. Se debe verificar que los tipos de las variables y expresiones sean correctos. Esto es:

- En la asignación los tipos de la variable y la expresión coinciden.
- En la instrucción **if** la condición es una expresión booleana.
- En el **while** la condición es booleana.

```
program ejemplo7;
var b : boolean;
    x : integer;
begin
   readln(b);
   readln(x);

   if b and x then
   begin
      writeln(x)
   end
   else
   begin
      while x * 4 do
      begin
         writeln(b)
      end
   end;
   x := (x + true) and (b or 8)
end.
```

Figure 7: Errores de Tipos

- El parámetro de **writeln** es una expresión entera.
- El parámetro de **readln** es una variable entera.
- Las sub-expresiones de una expresión tienen los tipos correctos de acuerdo al operador utilizado (ej. en e1 + e2, e1 y e2 deben ser enteros).

Por ejemplo, en la Figura 7 se muestra un programa que generaría los siguientes errores:

```
Expected: integer Actual: boolean
Expected: boolean Actual: integer
Expected: boolean Actual: integer
Expected: integer Actual: boolean
Expected: integer Actual: boolean
Expected: boolean Actual: integer
Expected: boolean Actual: integer
Expected: integer Actual: boolean
```

dado que se pasa una variable booleana al **readln**, uno de los operandos del **and** es de tipo entero, la condición del **while** es entera, se pasa una expresión booleana al **writeln**, uno de los operadores de la suma es booleano, uno de los operadores del **or** es entero, uno de los operadores del **and** es entero y se quiere asignar a una variable entera el resultado de una expresión booleana.

1.2 Optimizaciones

El compilador debe optimizar el programa utilizando las técnicas de *constant folding* y *dead code elimination*. Esto significa que se deben evaluar las expresiones constantes (ej. $(3 + 2 * 4)$ o **True and False**), reducir las identidades aritméticas y booleanas y eliminar trozos de programas a los que se asegura nunca se va a ingresar en la ejecución.

Estas optimizaciones se deben realizar en la función del módulo `Optimizer` que debe ser implementada como parte de la tarea:

```
optimize ::  Program -> Program
```

La función toma como entrada el árbol de sintaxis abstracta de un programa correcto y retorna el árbol de sintaxis abstracta de un programa equivalente optimizado.

Las optimizaciones de *constant folding* que se deben realizar en expresiones y sub-expresiones son:

- Si el o los operandos son todos constantes, evaluar la operación (ej. la expresión $(3 + 2 * 4)$ se transforma en 11).
- Si en una expresión entera o booleana al menos uno de sus operandos es constante y puede determinar su resultado, reducirla (ej. x **and False** se transforma en **False**, $z * 1$ se transforma en z, etc.).

Las optimizaciones no harán uso ni de asociatividad ni de conmutatividad de los operadores. Tampoco se realizará propagación de constantes, esto es sustituir una variable cuyo valor es constante por dicha constante.

Las optimizaciones de *dead code elimination*, que se deben realizar al código resultante de la optimización anterior, son:

- Si en un **if** la condición es constante, se sustituye por la rama correspondiente.
- Si en un **while** la condición es falsa, el mismo se elimina.

En la Figura 8 se muestra un ejemplo de programa que al ser optimizado resultaría en el programa de la Figura 9.

1.3 Generación de Código

Luego de superar de forma exitosa la fase de chequeos y de pasar por la fase de optimización, el compilador debe proceder a generar código que pueda ser ejecutado por una máquina. En este caso generaremos código de una máquina abstracta basada en stacks, que definiremos a continuación. La siguiente EBNF describe su sintaxis:

$\langle code \rangle \quad ::= [\langle instr \rangle \{ \text{';'} \langle instr \rangle \}]$

```
program ejemplo8;
var x : integer;
begin
    readln(x);
    x := 2 + x * 0;
    x := (3 + 2) * x;
    x := x + (0 div 3);
    if  (x = 38) and False then
    begin
        writeln(x)
    end
    else
    begin
        writeln(x+1)
    end;
    while ((2 = 3) or False) do
    begin
        writeln(x)
    end
end.
```

Figure 8: Programa a Optimizar

```
program ejemplo8;
var x : integer;
begin
    readln(x);
    x := 2;
    x := 5 * x;
    x := x;
    writeln(x+1)
end.
```

Figure 9: Programa Optimizado

⟨*instr*⟩ ::= 'push' ⟨*int*⟩
| 'neg' | 'add' | 'sub' | 'mul' | 'mod'
| 'cmp'
| 'jump' ⟨*int*⟩ | 'jmpz' ⟨*int*⟩
| 'load' ⟨*ident*⟩ | 'store' ⟨*ident*⟩
| 'read' | 'write'
| 'skip'

Notar que el código puede ser vacío.

La instrucción **push** agrega un entero en el tope del stack. Notar que el stack sólo puede contener enteros, que es el único tipo de datos que manipula este lenguaje. La instrucción **neg** quita el entero v que se encuentre en el tope del stack y lo sustituye por $-v$. Las instrucciones **add**, **sub**, **mul** y **mod**, quitan dos enteros del tope del stack e insertan el resultante de realizar la operación correspondiente. Por ejemplo al ejecutarse el código **push** 2; **push** 5; **sub** el stack queda con el entero 3 en su tope.

La instrucción **cmp** quita los enteros v_1 y v_2 del tope del stack, los compara e inserta 1 si $v_1 > v_2$, 0 si $v_1 = v_2$ o -1 si $v_1 < v_2$. Por ejemplo al ejecutarse el código **push** 2; **push** 5; **cmp** el stack queda con el entero 1 en su tope.

Las instrucciones de salto **jump** y **jmpz** mueven el *program counter* (PC) tantas instrucciones como el entero que se le pasa. Si el entero es positivo se mueve hacia adelante en la secuencia de instrucciones y si es negativo se mueve hacia atrás. La instrucción **jump** realiza un salto incondicional, es decir que siempre mueve el PC, mientras que **jmpz** quita el primer entero del stack y realiza el salto sólo si ese entero es 0. Por ejemplo el siguiente código agrega sólo un 2 en el tope del stack:
push 0; **jmpz** 2; **push** 1; **push** 2
mientras que el siguiente agrega un 1 y un 2:
push 8; **jmpz** 2; **push** 1; **push** 2

La máquina también maneja un ambiente de variables, que puede ser manipulado con las instrucciones **load** y **store**. Con la primera se obtiene el valor (entero) de una variable dada y se lo coloca en el tope del stack, mientras que con la segunda se puede asignar a una variable el valor que se encuentre en el tope del stack (este valor se quita del stack).

Las funciones de entrada y salida se realizan con las instrucciones **read** y **write**. La lectura se realiza con **read**, que lee un entero de la entrada y lo coloca en el tope del stack, y la escritura con **write**, que quita un entero del tope del stack y lo imprime.

Finalmente, la instrucción **skip** no tiene ningún efecto.

El módulo `MachineLang`, provisto en el archivo `MachineLang.hs`, contiene un tipo algebraico de datos `Code` que representa a los árboles de sintaxis abstracta de código de máquina en este lenguaje.

La generación de código será realizada por la función del módulo `Generator` que debe ser implementada como parte de la tarea:

```
generate ::  Program -> Code
```

La función toma como entrada el árbol de sintaxis abstracta de un programa MicroPascal correcto y retorna el árbol de sintaxis abstracta de un código equivalente en lenguaje de máquina.

Al realizarse la traducción se debe tener en cuenta que el lenguaje de máquina no tiene booleanos. Los mismos se pueden codificar con enteros, donde 0 es false y distinto de 0 (ej. 1) es true, teniendo que usar saltos condicionales para codificar a los operadores.

1.4 Intérprete

El código generado será interpretado por la función `interp`, declarada en el módulo `Interpreter`, que debe ser implementada como parte de la tarea. Se puede asumir que la función sólo interpreta código compilado, el cual se supone es correcto respecto a su estructura (todo jump salta a una dirección válida).

```
interp ::  Code -> Code -> Conf -> IO Conf
```

Esta función implementa a la máquina abstracta que interpreta al lenguaje. Recibe como parámetros la secuencia de instrucciones que son anteriores al PC (se pasan en orden inverso), la secuencia de instrucciones a partir del PC y la configuración inicial, donde una configuración consta de un par formado por el stack y el ambiente de variables. El resultado es la configuración final, con los cambios al stack y el ambiente que hayan realizado las instrucciones. La función es monádica (el resultado es una computación en la mónada IO) porque la interpretación de **read** y **write** requiere interactuar con la entrada y la salida respectivamente.

Veamos un ejemplo de cómo se comporta la interpretación de un código, siendo *amb* el ambiente de variables:

```
interp [] [PUSH 3, PUSH 4, ADD] ([],amb)
⇝ interp [PUSH 3] [PUSH 4, ADD] ([3],amb)
⇝ interp [PUSH 4, PUSH 3] [ADD] ([4,3],amb)
⇝ interp [ADD, PUSH 4, PUSH 3] [] ([7],amb)
⇝ return ([7],amb)
```

2 Se Pide

Además de esta letra el obligatorio contiene los siguientes archivos:

`Syntax.hs`	Módulo que contiene el parser y el AST.
`TypeChecker.hs`	Módulo de chequeos.
`Optimizer.hs`	Módulo de optimización.
`MachineLang.hs`	Módulo que contiene el AST del código de máquina.
`Generator.hs`	Módulo de generación de código.
`Interpreter.hs`	Módulo del intérprete.

`MicroPascal.hs` Programa Principal, importa los módulos anteriores y define un intérprete, que dado el *nombre* de un programa MicroPascal obtiene el programa de un archivo *nombre.mp*, chequea que sea válido y en caso de serlo, lo optimiza, genera el código de máquina y lo interpreta. En otro caso imprime los errores encontrados. El intérprete cuenta con dos flags que permiten visualizar información interna de los programas: `-o` despliega el AST del programa optimizado y `-m` despliega el AST del código de máquina generado.

`ejemplo`*i*`.mp` Programas MicroPascal usados como ejemplos en esta letra.

`ejemplo`*i*`.err` Mensajes de error impresos por el compilador en caso de que se encuentren errores en los chequeos.

La tarea consiste en modificar los archivos `TypeChecker.hs`, `Optimizer.hs`, `Generator.hs` e `Interpreter.hs`, implementando las funciones solicitadas, de manera que el intérprete se comporte como se describe en esta letra.

Los únicos archivos que se entregarán son `TypeChecker.hs`, `Optimizer.hs`, `Generator.hs` e `Interpreter.hs`. Dentro de ellos se pueden definir todas las funciones auxiliares que sean necesarias. No modificar ninguno de los demás archivos, dado que los mismos no serán entregados.