Laboratorio de Programación Funcional 2022
Compilador e Intérprete de MicroC

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

1 MicroC

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

⟨*program*⟩ ::= { (⟨*def*⟩ | ⟨*stmt*⟩) '`;`' }

⟨*def*⟩ ::= ⟨*type*⟩ ⟨*ident*⟩

⟨*type*⟩ ::= '`int`' | '`char`'

⟨*stmt*⟩ ::= ⟨*expr*⟩
| '`if`' '`(`' ⟨*expr*⟩ '`)`' ⟨*body*⟩ '`else`' ⟨*body*⟩
| '`while`' '`(`' ⟨*expr*⟩ '`)`' ⟨*body*⟩
| '`putchar`' '`(`' ⟨*expr*⟩ '`)`'

⟨*body*⟩ ::= '{' { ⟨*stmt*⟩ '`;`' } '}'

⟨*expr*⟩ ::= ⟨*ident*⟩ | ⟨*char*⟩ | ⟨*nat*⟩
| ⟨*ident*⟩ '`=`' ⟨*expr*⟩ | '`getchar`' '`(`' '`)`'
| ⟨*expr*⟩ ⟨*bop*⟩ ⟨*expr*⟩ | ⟨*uop*⟩ ⟨*expr*⟩

⟨*bop*⟩ ::= '`||`' | '`&&`' | '`==`' | '`<`' | '`+`' | '`-`' | '`*`' | '`/`' | '`%`'

⟨*uop*⟩ ::= '`!`' | '`-`'

Un programa consiste en una secuencia (posiblemente vacía) de declaraciones de variables e instrucciones.

```
int  x;
char c;
int  otra;
```

Figure 1: Programa con declaraciones

```
int  x;
char c;
4 + 5;
c = getchar();
x = 0;
x && (2 * 0);
```

Figure 2: Programa con expresiones

Al declarar una variable se especifica su tipo y su nombre. En MicroC hay dos tipos: **int** y **char**. Notar que, al igual que en C, no hay un tipo para los booleanos, los cuales se representan como enteros donde 0 es false y cualquier otro valor es true. En la Figura 1 se muestra un programa que solo tiene declaraciones de variables.

Las expresiones pueden ser variables (ej. x), literales caracteres, literales enteros, aplicación de un operador unario a una sub-expresión, aplicación de un operador binario a dos sub-expresiones. Los operadores unarios son la negación $-$ y el no lógico !. Los operadores binarios son los operadores lógicos || y &&, los operadores de comparación == y $<$, y los operadores enteros $+$, $-$, $*$, / y %. También son expresiones la asignación y la lectura de caracteres (**getchar**).

En MicroC las expresiones son también instrucciones. En la Figura 2 podemos ver un programa compuesto de expresiones.

Además de las expresiones se tienen instrucciones de: selección **if**$-$then$-$**else** (no se puede omitir el **else**), iteración condicional **while**, y de impresión de caracteres **putchar**. En la Figura 3 se muestra un programa que utiliza estas instrucciones.

Notar que no existe una sección especial del programa para las declaraciones de variables. Como se puede ver en el programa de la Figura 4, las variables se pueden introducir en cualquier punto de la secuencia de instrucciones, siempre y cuando sean utilizadas solo luego de ser introducidas.

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 MicroC y una función de parsing, que dada una cadena de caracteres con un programa MicroC retorna su árbol de sintaxis abstracta o los errores de sintaxis encontrados al intentar parsear:

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

```
int x;
x = 0;
while (x < 10){
  x = x + 1;
};
if (x == 10) {
  putchar('a');
}
else {

};
```

Figure 3: Instrucciones.

```
int x;
x = 0;
x = x + 2;
char c;
c = getchar();
x = c == 'c';
```

Figure 4: Instrucciones y Declaraciones.

```
int x;
int x;
z = x + w + y;
char x;
char w;
z = x + w + y;
```

Figure 5: Chequeo de Nombres

2 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.

2.1 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 anteriormente 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: x
Undefined: z
Undefined: w
Undefined: y
Duplicated definition: x
Undefined: z
Undefined: y
```

Notar que las repeticiones y el uso de nombres no definidos se listan a medida que se van encontrando en una recorrida de arriba hacia abajo. Las expresiones se recorren de izquierda a derecha.

2.2 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:

```
char x;
x = 2 + getchar();
while (x) { putchar(8 == x); };
```

Figure 6: Errores de Tipos

- En la asignación el tipo de la expresión coincide con el tipo de la variable.
- En las instrucciones **if** y **while** la condición es una expresión entera.
- El parámetro de **putchar** es una expresión de tipo caracter.
- Las sub-expresiones de una expresión tienen los tipos correctos de acuerdo al operador utilizado; es decir que los operadores aritméticos (+, −, ∗, / y %) y lógicos (!, || y &&) tienen operandos enteros, y los operandos de los operadores de comparación (== y <) deben ser ambos enteros o caracteres (ej. en e1 < e2, si e1 es entero e2 debe ser entero y si e1 es caracter e2 debe ser caracter).

Por ejemplo, en la Figura 6 se muestra un programa que generaría los siguientes errores:

```
Expected: int Actual: char
Expected: char Actual: int
Expected: int Actual: char
Expected: int Actual: char
Expected: char Actual: int
```

Que se originan respectivamente por:

- En la suma una de las subexpresiones es **char**.
- En la asignación el tipo de la expresión es **int**, pero la variable es **char**.
- La condición del **while** es **char**.
- En la igualdad el tipo de la sub-expresión de la derecha (**char**) no coincide con el de la izquierda (**int**).
- El argumento de **putchar** es de tipo **int**.

El orden en que se reportan los errores en su implementación debe ser el que damos en el ejemplo.

3 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 0 && 1), reducir neutros y nulos y eliminar

trozos de programas a los que se asegura que 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:

- Neutro a la izquierda para +, ∗, && y `||`. Ejemplo: 1 ∗ x se transforma en x.
- Neutro a la derecha para +, ∗, && y `||`. Ejemplo: x && 10 se transforma en x.
- Nulo a la izquierda para ∗, && y `||`. Ejemplo: `20 || x` se transforma en 20.
- Nulo a la derecha para ∗, && y `||`. Ejemplo: x ∗ 0 se transforma en 0.
- Si el o los operandos de una operación aritmética son todos constantes, evaluar la operación. Ejemplo: (13 / 6) se transforma en 2.

Las optimizaciones se realizan todas en ese orden primero para la sub-expresión izquierda (y recusivamente para sus sub-expresiones), luego para la derecha (y sus sub-expresiones) y finalmente para la expresión raíz, de manera de aprovechar la propagación de optimizaciones. No se hace uso ni de asociatividad ni de conmutatividad de los operadores. Tampoco se realizará propagación de constantes, esto es sustituir una variable por su valor.

Las optimizaciones de *dead code elimination*, que se deben realizar al código resultante de las optimizaciones anteriores, 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 7 se muestra un ejemplo de programa que al ser optimizado resultaría en el programa de la Figura 8.

4 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

```
int x;
int y;
x = (18 || 0) && 15;
if (x && 1) {
  y = 10 * 2;
}
else{
  y = 10 * 0;
};
while (0 && x) { y = y + 1; };
```

Figure 7: Programa a Optimizar

```
int x;
int y;
x = 15;
if (x) {
  y = 20;
}
else{
  y = 0;
};
```

Figure 8: Programa Optimizado

abstracta basada en stacks, que definiremos a continuación. La siguiente EBNF describe su sintaxis:

⟨*code*⟩ ::= [⟨*instr*⟩{';'⟨*instr*⟩}]

⟨*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** sustituye el entero v que se encuentre en el tope del stack por $-v$. Las instrucciones **add**, **sub**, **mul** y **mod**, quitan los dos enteros que están en el tope del stack e insertan en el tope 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 en el tope 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 caracter de la entrada y coloca el entero correspondiente a su código ASCII en el tope del stack, y la escritura con **write**, que quita el entero que está en el tope del stack, asume que representa un código ASCII, e imprime el caracter correspondiente.

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 MicroC 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 operadores booleanos. Al igual que en MicroC, los booleanos 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.

5 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 (en orden inverso) que son anteriores al *program counter* (PC), la secuencia de instrucciones a partir del PC y la configuración inicial de la máquina, 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 al 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 estándar 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)
```

6 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.

`MicroC.hs` Programa Principal, importa los módulos anteriores y define un intérprete, que dado el *nombre* de un programa MicroC obtiene el programa de un archivo *nombre.mc*, 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*`.mc` Programas MicroC 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.