Laboratorio de Programación Funcional 2017
Compilador de MiniHaskell

La tarea del laboratorio consiste en la implementación de un compilador que toma programas en una versión simplificada de Haskell, que llamaremos MiniHaskell, y genera programas equivalentes en C. Además de la generación de código C, el compilador deberá realizar las tareas de chequeo de buena formación de los programas fuente escritos en MiniHaskell.

1 MiniHaskell

La siguiente EBNF describe la sintaxis de MiniHaskell. Los no terminales son escritos entre paréntesis angulares, como ser ⟨*program*⟩, ⟨*expr*⟩, etc. Los terminales son escritos entre comillas simples. El no terminal ⟨*ident*⟩ representa un identificador válido e ⟨*int*⟩ un número entero no negativo.

⟨*program*⟩ ::= ⟨*fundefs*⟩ '`main`' '`=`' ⟨*expr*⟩

⟨*fundefs*⟩ ::= { ⟨*fundef*⟩ }

⟨*fundef*⟩ ::= ⟨*funtype*⟩ ⟨*funeq*⟩

⟨*funtype*⟩ ::= ⟨*ident*⟩ '`::`' '`(`'⟨*type*⟩{'`,`'⟨*type*⟩}'`)`' '`->`' ⟨*type*⟩

⟨*funeq*⟩ ::= ⟨*ident*⟩ '`(`'⟨*ident*⟩{'`,`'⟨*ident*⟩}'`)`' '`=`' ⟨*expr*⟩

⟨*type*⟩ ::= '`Int`' | '`Bool`'

⟨*expr*⟩ ::= ⟨*ident*⟩ {'`(`'⟨*expr*⟩ {'`,`'⟨*expr*⟩}'`)`'} | '`True` | `False`' | ⟨*int*⟩ | ⟨*expr*⟩ ⟨*op*⟩ ⟨*expr*⟩ | '`if`' ⟨*expr*⟩ '`then`' ⟨*expr*⟩ '`else`' ⟨*expr*⟩ | '`let`' ⟨*ident*⟩ '`::`' ⟨*type*⟩ '`=`' ⟨*expr*⟩ '`in`' ⟨*expr*⟩

⟨*op*⟩ ::= '`+`' | '`-`' | '`*`' | '`div`' | '`==`' | '`/=`' | '`<`' | '`>`' | '`<=`' | '`>=`'

Un programa está formado por una lista (posiblemente vacía) de declaraciones de funciones y una definición `main` dada por una expresión (`main` representa el programa principal). En la Figura 1 se muestra el ejemplo de un programa minimal en el que no se declaran funciones.

```
main = 24 + 2
```

Figure 1: Programa sin declaraciones

```
cuad :: (Int) -> Int
cuad(x) = x * x

main = 24 + cuad(2)
```

Figure 2: Declaración de función

Una función se declara mediante la especificación de su tipo y su ecuación; ambas componentes contienen el nombre de la función, el cual debe ser el mismo (esto es chequeado por el parser). Los valores manipulados por MiniHaskell son de dos tipos: `Int` y `Bool`. A diferencia de Haskell, MiniHaskell no es un lenguaje de alto orden por lo que no se pueden pasar funciones como parámetro ni retornar funciones como resultado.

En la Figura 2 se puede ver un programa en el que se declara una única función de enteros en enteros. Notar que en el tipo de las funciones el dominio siempre se escribe entre paréntesis, incluso en el caso en que tenga un único parámetro. El mismo tratamiento se tiene con los parámetros en las ecuaciones.

Los cuerpos de las funciones están dados por expresiones, que pueden contener variables, aplicaciones de funciones a argumentos dados por expresiones, literales booleanos, literales enteros, aplicaciones de operadores binarios a dos subexpresiones, expresiones `if` o expresiones `let`. La variable (local) de una expresión `let` tiene anotado el tipo.

Las Figuras 3 y 4 muestran ejemplos de programa que utilizan varios de estos tipos de expresiones. Notar que en MiniHaskell hay operadores relacionales (como `==`, `<`), pero no así operadores lógicos (como `and`, `or`). Como es habitual, los operadores relacionales comparan valores del mismo tipo y retornan un booleano. El resto de los operadores binarios operan sólo entre enteros.

Como es de esperar, MiniHaskell permite definir funciones recursivas. La Figura 5 muestra un programa que define la función factorial.

El módulo `Syntax`, provisto en el archivo `Syntax.hs`, contiene:

- un tipo algebraico de datos `Program` que representa a los árboles de sintaxis

```
foo :: (Int,Bool) -> Int
foo(x,y) = if y then x * x else x + 2

main = 24 + foo(2,True) + foo(3,False)
```

Figure 3: Uso de varios tipos de expresión

```
foo :: (Int,Int) -> Int
foo(x,y) = (let x :: Int = y in x)
              + (let y :: Int = x + y in y)

main = foo(2,4)
```

Figure 4: Uso de `let`

```
fact :: (Int) -> Int
fact (x) = if x==0 then 1 else x*fact(x-1)

main = fact(4)
```

Figure 5: Definición de función recursiva

abstracta de programas MiniHaskell, y

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

  `parser :: String -> Either ParseError Program`

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 `Checker` que debe ser implementada como parte de la tarea:

```
checkProgram :: Program -> Checked
```

donde

```
data Checked = Ok | Wrong [Error]
```

La función toma como entrada el árbol de sintaxis abstracta de un programa y retorna `Ok` en caso de no encontrar errores, o la lista de errores encontrados en otro caso.

La fase de chequeos se realiza en varias etapas. En caso de que se detecten errores al finalizar una etapa entonces se aborta (no se continúa con las siguientes etapas) y se reportan los errores encontrados. A continuación se detallan las distintas etapas.

Repetición de nombres En primera instancia se chequea que el programa no contenga nombres repetidos. Primero se chequea que no hayan múltiples declaraciones de una misma función y luego a continuación se chequea que no

```
f :: (Int,Int,Int,Int) -> Int
f (x,y,x,x) = x

g :: (Int,Int,Int,Int,Int) -> Int
g (x,y,z,z,x) = x + y + z

f :: (Int) -> Int
f (x) = x

f :: (Int,Int) -> Int
f (s,s) = s

g :: (Int) -> Int
g (x) = x * z

main = f (2)
```

Figure 6: Programa con nombres repetidos

haya nombres de parámetros repetidos dentro de cada declaración de función. Por ejemplo, en el programa de la Figura 6 se detectan los siguientes errores:

```
Duplicated declaration: f
Duplicated declaration: g
Duplicated declaration: f
Duplicated declaration: x
Duplicated declaration: x
Duplicated declaration: x
Duplicated declaration: z
Duplicated declaration: s
```

Notar que se reporta cada declaración de un nombre repetido y en el orden de aparición.

Número de parámetros Se chequea que el número de parámetros especificados en la ecuación de una función es igual al número de parámetros del dominio declarado en la signatura de la función. En el caso del programa de la Figura 7 se reporta el siguiente error:

```
The number of arguments in the definition of f doesn't match the
signature (2 vs 1)
```

Nombres no declarados Se chequea que no se usen nombres que no hayan sido declarados. Por ejemplo, en el programa de la Figura 8 se reportan los siguientes errores:

```
f :: (Int) -> Int
f (x,y) = x + y

main = f(4)
```

Figure 7: Diferencia en número de parámetros

```
Undefined: z
Undefined: g
Undefined: s
Undefined: h
Undefined: x
```

Notar que se reporta cada uso de un nombre no declarado y en el orden de aparición.

```
f :: (Int) -> Int
f (x) = x + z + g(s)

main = h(x)
```

Figure 8: Uso de nombres no declarados

Chequeo de Tipos Se debe chequear que las expresiones que ocurren en el programa tienen el tipo correcto. Esta tarea se realiza utilizando la signatura de las funciones declaradas y el tipo de las variables que existen en el scope de la expresión. Para efectuar el chequeo de tipos es común que la información de las variables en el scope se lleve en un *ambiente*, que es una tabla conteniendo el nombre de cada variable y su tipo.

A continuación describimos las reglas para tipar las expresiones en el contexto de un ambiente de variables `env`. La descripción es dada en términos de la sintaxis del lenguaje.

- Una variable tiene el tipo con el que aparece en el ambiente `env`.
- Un literal entrero tiene tipo `Int`.
- Un literal booleano tiene tipo `Bool`.
- El tipo de una expresión infija `e op e'` depende del operador `op`.
  - En el caso de los operadores aritméticos las subexpresiones `e` y `e'` deben ser de tipo entero, el resultado es entero.
  - En el caso de los operadores relacionales las subexpresiones `e` y `e'` deben ser del mismo tipo y el resultado es de tipo `Bool`.

```
f :: (Int, Int) -> Int
f (x,y) = if x then x+y else x==True

main = False == f (True,2+(True*4))
```

Figure 9: Errores de Tipo

- El tipo de una expresión `if b then e else e'` es `t` si `e` y `e'` son ambas de tipo `t`; la expresión `b` debe ser de tipo `Bool`.
- El tipo de una expresión `let x :: t = e in e'` es el tipo de `e'`. La expresión `e` tiene que ser de tipo `t` (que es el tipo de la variable local `x`). La expresión `e'` se debe tipar en el ambiente `env` extendido con la información del tipo de `x`.
- El tipo de una aplicación `f (e1,...,en)` es `t` si `f :: (t1,...,tn) -> t`. Se debe chequear que cada argumento `ei` tiene tipo `ti`.

En la Figura 9 se muestra un programa que produce los siguientes errores:

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

Los errores de tipo se reportan a medida que van apareciendo. En cada error de tipo se informa el tipo que se esperaba y el tipo que en realidad se tiene.

2.1 Generación de Código

Luego de superar de forma exitosa la fase de chequeos, el compilador debe proceder a generar código en el lenguaje objeto que es C. El programa generado en C debe estar en condiciones de poderse compilar y ejecutar.

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

```
genProgram :: Program -> String
```

El no tener funciones de alto orden en MiniHaskell tiene como beneficio que la traducción a C sea bastante directa. Para realizar la traducción se tendán en cuenta los siguientes detalles:

- En C no hay booleanos. Se pueden codificar con enteros, donde 0 es false y distinto de 0 (ej. 1) es true.

```
#include <stdio.h>
int main() {
printf("%d\n",(24 + 2)); }
```

Figure 10: Código generado a partir de Fig. 1

```
#include <stdio.h>
int _cuad(int _x){
return ((_x * _x)); };
int main() {
printf("%d\n",(24 + _cuad(2))); }
```

Figure 11: Código generado a partir de Fig. 2

- Para evitar posibles conflictos de nombres con palabras reservadas de C, los identificadores del programa MiniHaskell son traducidos prefijándoles el caracter '_'. De esta forma, por ejemplo, el identificador `x` se traduce como `_x`.
- En la traducción se tendrá en cuenta que C permite definiciones de funciones, incluso por recursión.

Teniendo lo anterior en cuenta, en las Figuras 10, 11 y 12 se muestran los códigos generados a partir de los programas de las Figuras 1, 2 y 3, respectivamente.

El caso que precisa más cuidado es cuando ocurre una expresión `let`. La traducción de `let x :: t = e in e'` resulta en una llamada a una función C de nombre `_let`n (con n natural) aplicada a la traducción de la expresión `e`. La función `_let`n se debe definir y es tal que tiene un parámetro nominal `_x` de tipo la traducción de `t` y como cuerpo la traducción de `e'`. La Figura 13 muestra la traducción del programa de la Figura 4 en el cual se utilizan `let`s.

La Figura 14 muestra la traducción del programa de la Figura 5 el cual contiene la definición de una función recursiva.

```
#include <stdio.h>
int _foo(int _x,int _y){
return (_y?(_x * _x):(_x + 2)); };
int main() {
printf("%d\n",((24 + _foo(2,1)) + _foo(3,0))); }
```

Figure 12: Código generado a partir de Fig. 3

```
#include <stdio.h>
int _foo(int _x,int _y){
int _let0(int _x){
return (_x); };
int _let1(int _y){
return (_y); };
return ((_let0(_y) + _let1((_x + _y)))); };
int main() {
printf("%d\n",_foo(2,4)); }
```

Figure 13: Código generado a partir de Fig. 4 (traducción de `lets`)

```
#include <stdio.h>
int _fact(int _x){
return ((_x==0)?1:(_x * _fact((_x - 1)))); };
int main() {
printf("%d\n",_fact(4)); }
```

Figure 14: Código generado a partir de Fig. 5 (traducción de recursión)

3 Archivos

Además de esta letra el obligatorio contiene los siguientes archivos:

`Syntax.hs` Módulo que contiene el parser y el AST.

`Checker.hs` Módulo de chequeos.

`Generator.hs` Módulo de generación de código.

`Compiler.hs` Programa Principal, importa los tres módulos anteriores y define una función de compilación, tal que dado el *nombre* de un programa MiniHaskell obtiene el programa de un archivo *nombre.mhs*, chequea que sea válido y en caso de serlo genera un archivo *nombre.c* con el código C correspondiente. En otro caso imprime los errores encontrados.

`ejemplo`*i*`.mhs` Programas MiniHaskell usados como ejemplos en esta letra.

`ejemplo`*i*`.c` Programas C generados en caso de que no se encuentren errores en los chequeos.

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

4 Se pide

La tarea consiste en modificar los archivos `Checker.hs` y `Generator.hs`, implementando las funciones solicitadas, de manera que el compilador se comporte como se describe en esta letra.

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