Laboratorio de Programación Funcional 2023
Compilador de FUN

La tarea de laboratorio consiste en la implementación de un compilador para un lenguaje funcional de primer orden que llamaremos FUN. El compilador toma como entrada programas fuente escritos en FUN y genera como salida programas equivalentes en el lenguaje C, luego de realizar chequeos de buena formación y posibles optimizaciones sobre los programas fuente.

1 El lenguaje FUN

La siguiente sintaxis describe el lenguaje FUN en EBNF[1]. Los no terminales de la gramática son escritos entre paréntesis angulares, como ser ⟨*prog*⟩, ⟨*type*⟩, etc. Los símbolos terminales son escritos entre comillas simples. El no terminal ⟨*ident*⟩ representa un identificador válido e ⟨*int*⟩ un número entero no negativo.

⟨*prog*⟩ ::= ⟨*fundefs*⟩ '`main`' '`=`' ⟨*exp*⟩

⟨*fundefs*⟩ ::= { ⟨*fundef*⟩ }

⟨*fundef*⟩ ::= ⟨*funtype*⟩ ⟨*funeq*⟩

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

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

⟨*type*⟩ ::= '`Int`' | '`Bool`'

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

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

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

[1] extended BNF

```
main = 23 + 4
```

Figure 1: Programa sin declaraciones

```
double :: (Int) -> Int
double(x) = 2 * x

main = 23 + double(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). Cada función tiene una única ecuación. FUN manipula valores de dos tipos: `Int` y `Bool`. Este lenguaje es de primer orden (y no de alto orden como Haskell) por lo que no se pueden pasar funciones como parámetro ni retornar funciones como resultado.

La Figura 2 muestra 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 literales enteros y booleanos, variables, aplicaciones de funciones a argumentos dados por expresiones, aplicaciones de operadores binarios, expresiones `if` o expresiones `let`. La variable (local) de una expresión `let` tiene anotado su tipo.

Las Figuras 3 y 4 muestran ejemplos de programas que utilizan varios de estos tipos de expresiones. Notar que en FUN hay operadores relacionales (como `==`, `<`), pero no así operadores lógicos (como `and` u `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 era de esperar, FUN permite la definición de funciones recursivas. La Figura 5 muestra un programa que define el clásico ejemplo de la función factorial.

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

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

main = 23 + 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

- un tipo algebraico de datos `Program` que representa a los árboles de sintaxis abstracta de programas FUN, y
- una función de parsing, que dada una cadena de caracteres con un programa FUN retorna su árbol de sintaxis abstracta o los errores de sintaxis encontrados al intentar parsear:

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

2 Chequeos

El compilador de FUN 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.

```
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

2.1 Repetición de nombres

En primera instancia se chequea que el programa no contenga nombres repetidos. La verificación se debe realizar en este orden: primero se chequea que no hayan múltiples declaraciones de una misma función y luego a continuación se chequea que no 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: f
Duplicated declaration: g
Duplicated declaration: x
Duplicated declaration: x
Duplicated declaration: z
Duplicated declaration: x
Duplicated declaration: s
```

Notar que los nombres repetidos se reportan en el orden que van ocurriendo.

2.2 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)
```

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

main = f(4)
```

Figure 7: Diferencia en número de parámetros

2.3 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:

```
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

2.4 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`.

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

main = False == f (True,2+(True*4),5)
```

Figure 9: Errores de Tipo

- 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`.
- 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,...,ek)` es `t` si `f :: (t1,...,tn) -> t`. Se debe chequear que la cantidad `k` de argumentos pasados coincida con el número `n` de parámetros declarados en el tipo de la función y (independiente de si `k` y `n` coinciden) chequear que cada argumento `ei` tiene tipo `ti`. Se debe chequear el tipo de `min(k,n)` argumentos. O sea, si `k > n` entonces sólo se van a chequear los primeros `n` argumentos y los restantes se ignoran. Si `k <= n` se chequean los `k` argumentos que se pasaron.

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

```
Expected: Bool Actual: Int
Expected: Int Actual: Bool
Expected: Int Actual: Bool
Expected: Bool Actual: Int
The number of arguments in the application of: f doesn't match the signature (3 vs 2)
Expected: Int Actual: Bool
Expected: Int Actual: Bool
```

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.

3 Eliminación de lets

Luego de superar de forma exitosa la fase de chequeos y antes de generar código el compilador va a aplicar una optimización sobre el programa fuente en la que se van a eliminar aquellos `let`s en los que se liga una variable a un literal. Concretamente, todo `let` de la forma `let x :: t = c in e`, donde `c` es un *literal* entero o booleano, va a ser transformado en una nueva expresión, que denotamos como `e[c/x]`, que resulta de sustituir por el literal `c` todas las *ocurrencias libres* de la variable `x` en la expresión `e`.

Una variable `x` se dice que ocurre *libre* en una expresión `e` cuando no aparece en el cuerpo de un `let` que liga esa misma variable. Caso contrario se dice que la variable ocurre *ligada*. Por ejemplo, `x` ocurre libre en la expresión `x + 2`. En cambio, en la expresión `let x :: Int = 4 * x in x + 2` la variable `x` tiene una ocurrencia libre (la que aparece en la expresión `4 * x`) y una ligada (la que aparece en la expresión `x + 2`).

En los siguientes ejemplos escribiremos `e` $\Longrightarrow$ `e'` para significar que `e` se transforma en `e'` y eliminaremos la anotación del tipo de la variable ligada de un `let` con el fin de mejorar la lectura.

```
let x = 3 in x + 2 * x                ⟹  3 + 2 * 3
let x = 3 * 4 in x + 2                ⟹  let x = 3 * 4 in x + 2
let x = 3 in let x = 4 in x + 2       ⟹  let x = 4 in x + 2
                                      ⟹  4 + 2
let x = 3 in let y = x in y + 2       ⟹  let y = 3 in y + 2
                                      ⟹  3 + 2
let x = 3 in let x = 4 + x in x + 2   ⟹  let x = 4 + 3 in x + 2
```

Hay casos en que la eliminación de un `let` dentro de la expresión ligada puede generar una oportunidad de eliminación sobre el `let` externo. Por ejemplo,

```
let x = (let y = 4 in y) in x + 2  ⟹  let x = 4 in x + 2
                                   ⟹  4 + 2
```

En cambio, hay otros casos en que la eliminación de un `let` interno puede que no tenga efecto en la eliminación del `let` externo.

```
let x = 5 + (let y = 4 in y) in x + 2  ⟹  let x = 5 + 4 in x + 2
let x = (let y = 4 in y + 3) in x + 2  ⟹  let x = 4 + 3 in x + 2
```

La implementación de esta optimización se debe realizar mediante la definición de la función

```
letElimP :: Program -> Program
```

que está contenida en el módulo `LetElim`. Esta función toma como entrada el árbol de sintaxis abstracta de un programa y retorna el árbol de sintaxis abstracta equivalente resultante de aplicar la optimización.

La implementación de `letElim` se apoya en una función

```
subst :: Name -> Expr -> Expr -> Expr
```

que también está contenida en el módulo `LetElim` y que opera sobre el árbol de sintaxis abstracta de expresiones. La invocación `subst x e1 e2` implementa la sustitución de las ocurrencias libres de `x` en `e2` por `e1` (lo que antes denotamos como `e2[e1/x]`).

Como parte de la tarea se deben implementar `letElimP` y `subst`.

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 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 hace que la traducción de FUN a C sea bastante directa. Para realizar la traducción se tendán en cuenta los siguientes detalles:

- C no tiene booleanos, pero se pueden codificar con enteros, donde 0 es false y distinto de 0 (ej. 1) es true.
- Para evitar posibles conflictos de nombres con palabras reservadas de C, los identificadores del programa FUN debe traducirse prefijándoles el caracter '_'. 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. Notar que en C las funciones deben ser definidas antes de su invocación, por lo tanto asumiremos (pero no lo chequearemos) que las funciones cumplen esta restricción.

Teniendo en cuenta lo anterior, 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`

```
#include <stdio.h>
int main() {
printf("%d\n",(23 + 4)); }
```

Figure 10: Código generado a partir de Fig. 1

```
#include <stdio.h>
int _double(int _x){
return ((2 * _x)); };
int main() {
printf("%d\n",(23 + _double(2))); }
```

Figure 11: Código generado a partir de Fig. 2

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.

5 Archivos

Además de esta letra el obligatorio contiene los siguientes archivos:

`Syntax.hs` Módulo que contiene el parser, el AST y pretty printing.

`Checker.hs` Módulo de chequeos.

`LetElim.hs` Módulo de optimización.

`Generator.hs` Módulo de generación de código C.

`Compiler.hs` Programa Principal.
Importa los cuatro módulos anteriores y define una función de compilación, tal que dado el *nombre* de un programa FUN obtiene el programa de un archivo *nombre.fun*, chequea que sea válido y en caso de serlo, lo

```
#include <stdio.h>
int _foo(int _x,int _b){
return (_b?(_x * _x):(_x + 2)); };
int main() {
printf("%d\n",((23 + _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)

optimiza (cuando la fase de optimización se encuentra habilitada) y genera un archivo de salida con el código C correspondiente. En caso que el programa no pase los chequeos imprime los errores encontrados.

El compilador cuenta con dos flags:

- una flag `-o` que habilita la fase de optimización.
- una flag `-p` que permite desplegar el AST del programa fuente optimizado (sólo tiene efecto cuando `-o` está habilitada).

Cuando la flag `-o` está habilitada se genera el archivo *nombre_opt.c* conteniendo el código C del programa optimizado. Cuando la flag no está habilitada se genera un archivo *nombre.c*.

`tests` Directorio conteniendo ejemplos de programas FUN:

`ejemplo`*i*`.fun` Programas FUN, algunos de los cuales son los usados como ejemplos en esta letra.

`ejemplo`*i*`.c` Programas C generados en caso de que no se encuentren errores en los chequeos (compilados sin la optimización habilitada).

`ejemplo`*i*`_opt.c` Programas C generados en caso de que no se encuentren errores en los chequeos (compilados con la optimización habilitada).

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

6 Se pide

La tarea consiste en modificar los archivos

- `Checker.hs`
- `LetElim.hs`
- `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 los mencionados (`Checker.hs`, `LetElim.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.

En el módulo `Checker.hs` **no** se debe modificar la definición de los tipos `Checked` y `Error` ni la instancia de `Show` para `Error`.