TIGER

Proyecto de Compilación

2013 – 2014

Colectivo de Compilación. MATCOM. UH. 2013-2014.

Tabla de Contenidos

TIGER: Proyecto de Compilación

PARTE I: GENERALES DEL PROYECTO

La evaluación de la asignatura Complementos de Compilación, inscrita en el programa del 4to año de la Licenciatura en Ciencia de la Computación de la Facultad de Matemática y Computación de la Universidad de La Habana, consiste este curso en la implementación de un compilador completamente funcional para el lenguaje **TIGER**.

TIGER es un pequeño lenguaje imperativo con variables enteras y de cadenas de caracteres, arreglos, records y funciones anidadas, cuya sintaxis se asemeja a la de algunos lenguajes funcionales.

ATENCIÓN: Si Ud. desea realizar otro proyecto como evaluación, o implementar el compilador de **TIGER** utilizando algún lenguaje o herramienta que no se mencione en este documento, Ud. debe estar previamente autorizado por los profesores de la asignatura.

SOBRE EL LENGUAJE TIGER

Ud. podrá encontrar la especificación formal del lenguaje **TIGER** en el documento *Tiger Language Reference Manual*, que se distribuye junto con el presente texto.

Para implementar el compilador de **TIGER**, además de la información que encontrará en el documento anterior, Ud. deberá tener en cuenta un conjunto de aclaraciones y modificaciones que el Colectivo de la Asignatura considera pertinente hacer a la especificación del lenguaje, y que aparecen recogidas más adelante en la PARTE II: TIGER - ACLARACIONES Y MODIFICACIONES.

SOBRE EL COMPILADOR DE TIGER

El compilador de **TIGER** debe ser un archivo ejecutable de consola de nombre **`tiger`** (con la extensión apropiada, por ejemplo: **`.exe`**) que reciba como primer y único argumento la ruta –relativa o absoluta– a un archivo de texto plano con extensión **`.tig`**, el cual contendrá el código del programa a compilar.

En caso de que ocurran errores durante la operación del compilador, **`tiger`** deberá terminar con código de salida (*exit code*) **`1`** y reportar a la salida estándar (*standard output stream*) lo que sigue…

```
<línea_con_nombre_y_versión_del_compilador>
<línea_con_copyright_para_el_compilador>

<línea_de_error>1
…
<línea_de_error>n
```

… donde `<línea_de_error>`$_i$ tiene el siguiente formato:

```
(<línea>,<columna>): <texto_del_error>
```

Los campos `<línea>` y `<columna>` indican la ubicación del error en el fichero **`.tig`** procesado. En caso de que la naturaleza del error sea tal que no pueda asociárselo a una línea y columna en el archivo de código fuente, el valor de dichos campos debe ser **`0`**.

Ejemplos:

```
D:\>tiger.exe C:\missing.tig
Tiger Compiler version 1.0
Copyright (C) 2013-2014 John Doe & Jane Doe

(0,0): File 'C:\missing.tig' cannot be found.

D:\>
```

… código de salida de **`tiger.exe`**: **`1`**

```
D:\>tiger.exe invalid.tig
Tiger Compiler version 1.0
Copyright (C) 2013-2014 John Doe & Jane Doe

(22,4): A value of type 'int' cannot be assigned to variable 'i', of type 'string'.
(35,18): Function 'facterial' cannot be found in current context.

D:\>
```

… código de salida de **`tiger.exe`**: **`1`**

En caso de que no ocurran errores durante la operación del compilador, **`tiger`** deberá terminar con código de salida **`0`**, generar (o sobrescribir si ya existe) un archivo ejecutable de consola (con la extensión apropiada, por ejemplo: **`.exe`**) en la misma carpeta del archivo **`.tig`** procesado y con el mismo nombre que éste, y reportar a la salida estándar solamente lo siguiente:

```
<línea_con_nombre_y_versión_del_compilador>
<línea_con_copyright_para_el_compilador>
```

Ejemplo:

```
D:\>tiger.exe "C:\Tiger Files\correct.tig"
Tiger Compiler version 1.0
Copyright (C) 2013-2014 John Doe & Jane Doe

D:\>
```

… código de salida de **`tiger.exe`**: **`0`**
… y se crea (o sobrescribe) el archivo **`C:\Tiger Files\correct.exe`**.

ATENCIÓN: No es necesario que Ud. implemente un programa con interfaz gráfica para su compilador.

SOBRE LOS EJECUTABLES GENERADOS POR EL COMPILADOR DE TIGER

Los programas generados por el compilador de **TIGER** deben ser ejecutables de consola.

Si durante la ejecución de un programa generado por el compilador de **TIGER** ocurre algún error inesperado, el programa debe terminar con código de salida **1** y reportar el error a la salida estándar de error (*standard error output*).

Ejemplo:

Al inicio del programa **`nullpointer.exe`**, generado por el compilador de **TIGER**, se intenta acceder al valor de cierto campo de un record a través de una variable con valor `nil`, por tanto, ocurre un error que es reportado a la salida estándar de error, y se termina con código de salida **1**. (El error aparece en este ejemplo en amarillo simplemente para indicar que ha sido reportado a la salida estándar de error, y no a la salida estándar.)

```
D:\>nullpointer.exe
Exception of type 'NullReferenceException' was thrown.

D:\>
```

… código de salida de **`nullpointer.exe`**: **1**

Si durante la ejecución de un programa generado por el compilador de **TIGER** no ocurre error, el programa debe terminar con código de salida **0**.

Ejemplo:

El programa **`factorial_10.exe`**, generado por el compilador de **TIGER**, computa satisfactoriamente y escribe a la salida estándar el valor de `10!`, para luego terminar con código de salida **0**.

```
D:\>factorial_10.exe
3628800

D:\>
```

… código de salida de **`factorial_10.exe`**: **0**

SOBRE LA IMPLEMENTACIÓN DEL COMPILADOR DE TIGER

Para la implementación del compilador Ud. debe utilizar una herramienta generadora de analizadores lexicográficos y sintácticos. Puede utilizar la que sea de su preferencia. La recomendación del Colectivo de la Asignatura es utilizar **`Antlr 3.4`**.

El proyecto puede ser desarrollado en los lenguajes **`C#`**, **`Java`**, **`Python`** o **`C/C++`** en el IDE de su preferencia. El producto final debe ser un compilador en forma de archivo ejecutable de consola (**`.exe`**, **`.class`**, **`.jar`** o **`.pyc`**) de nombre **`tiger`**. La recomendación del Colectivo de la Asignatura es utilizar **`C#`** como lenguaje de desarrollo del compilador.

El compilador debe ser capaz de procesar archivos **`.tig`** y construir el ejecutable de consola correspondiente (con la extensión apropiada: **`.exe`**, **`.class`**, **`.jar`** o **`.pyc`**) generando código en alguno de los siguientes lenguajes: **`IL`** (.NET), **`Java bytecode`**, **`Python bytecode`**, o **`C`** (y en este último caso se puede delegar en el compilador de **C** la generación del ejecutable). La recomendación del Colectivo de la Asignatura es generar ejecutables **`.exe`** con código **`IL`** (.NET).

SOBRE LOS EQUIPOS DE DESARROLLO

Para desarrollar el compilador del lenguaje **TIGER** se podrá trabajar de manera individual o en equipos de **2** integrantes.

SOBRE LOS MATERIALES A ENTREGAR

Para la evaluación del proyecto Ud. debe entregar su **COMPILADOR** en estado totalmente funcional, acompañado de **TODO EL CÓDIGO** a partir del cual fue generado y de los **EJEMPLOS DE PRUEBA** (ficheros **`.tig`**) que Ud. haya utilizado en la fase de *testing*.

Ud. entregará además un **INFORME** en formato **PDF** que resuma de manera organizada y comprensible la arquitectura e implementación de su compilador. El documento **NO** debe exceder las **5** cuartillas. En él explicará en más detalle su solución a los problemas que, durante la implementación de cada una de las fases del proceso de compilación, hayan requerido de Ud. especial atención.

La entrega de los materiales se realizará en un archivo **`ZIP`** en cuyo interior habrá 4 carpetas con el nombre y los contenidos que aparecen a continuación:

Nombre de la Carpeta:	Contenido de la Carpeta:
COMPILADOR	Compilador **`tiger`** y otros archivos de los que él dependa.
CODIGO	Código completo del proyecto (incluidos *lexer* y *parser*).
EJEMPLOS	Ficheros **`.tig`** de prueba que Ud. haya utilizado, válidos o no.
INFORME	Informe del proyecto en formato **PDF**.

El nombre completo del archivo **`ZIP`** debe tener un formato similar al del ejemplo siguiente:

`TIGER-JohnDoe(C411)-JaneDoe(C412).zip`

ATENCIÓN: No se aceptarán trabajos que no respeten los requisitos aquí plasmados con relación al archivo **`ZIP`** que se solicita.

SOBRE LA FECHA DE ENTREGA

Usted deberá entregar todos los materiales de su proyecto antes de la fecha siguiente:

lunes, 10 de marzo de 2014

ATENCIÓN: NO SE ADMITIRÁ NINGÚN MATERIAL PASADA LA FECHA DE ENTREGA.

PARTE II: TIGER - ACLARACIONES Y MODIFICACIONES

A continuación se plasman algunas aclaraciones y modificaciones que el Colectivo de la Asignatura ha considerado pertinente hacer a la especificación del lenguaje **TIGER** según aparece en el documento *Tiger Language Reference Manual*. Usted debe tenerlas en cuenta a la hora de implementar su compilador.

ACLARACIONES Y MODIFICACIONES SOBRE LO LEXICOGRÁFICO

ACLARACIÓN/MODIFICACIÓN 1

La especificación de **TIGER** plantea que los *whitespaces* y los *comments* deben ser excluidos de la secuencia de *tokens*. Los caracteres que determinarán un *whitespace* son única y exclusivamente los siguientes: *space*, *horizontal tab*, *line feed* y *carriage return*.

ACLARACIÓN/MODIFICACIÓN 2

Con respecto a los literales de cadenas de caracteres, la especificación del lenguaje **TIGER** dice:

"A string constant is a sequence of zero or more printable characters, spaces, or escape sequences surrounded by double quotes."

Un *printable character* es un caracter cuyo código ASCII está en el rango [32, 126]. El *space* es el carácter cuyo código ASCII es el 32; por tanto, el *space* es también un *printable character*. Con relación a las *escape sequences*, las que Ud. debe implementar son única y exclusivamente las siguientes:

`\n`	*Line feed*
`\r`	*Carriage return*
`\t`	*Horizontal tab*
`\"`	*Double quote*
`\\`	*Backslash*
`\ddd`	*The character with ASCII code* `ddd` *(three decimal digits)*
`\…\`	*Any sequence of whitespace characters (spaces, horizontal tabs, line feeds and carriage returns) surrounded by* \s *is ignored. This allows string constants to span multiple lines by ending and starting each with a backslash.*

La única manera de escribir los caracteres *line feed*, *carriage return*, *horizontal tab*, *double quote* y *backslash* dentro de un literal de cadena es a través del uso de *escape sequences*.

En el caso de la *escape sequence* `\ddd`, si los dígitos `ddd` no determinan un entero en el rango [0, 127] dicha *scape sequence* se considerará inválida y, por tanto, será inválido el literal de cadena de caracteres que la contenga. La invalidez de dichos literales de cadenas de caracteres debe ser identificada durante la fase de análisis lexicográfico.

Considérese el siguiente programa para una mejor comprensión de la *escape sequence* `\…\`.

```
/*
Este programa es válido e imprime la siguiente cadena (en una sola línea):
Tigers are solitary, nocturnal hunters, preying on medium-sized mammals.
*/

let
      var info : string := "Tigers are solitary, nocturnal hunters, \
                           \preying on medium-sized mammals."
in
      print(info)
end
```

Finalmente, observe que los literales de cadenas de caracteres en **TIGER** no pueden contener caracteres como los siguientes: `ñ`, `á`, `ü`, `È`, etc. Dichos caracteres no están en el alfabeto ASCII.

ACLARACIONES Y MODIFICACIONES SOBRE LO SEMÁNTICO

ACLARACIÓN/MODIFICACIÓN 1

Se eliminan de la ***Biblioteca Estándar de TIGER*** las funciones `getchar()` y `flush()`.

Se adicionan a la ***Biblioteca Estándar de TIGER*** las funciones `getline():string`, que se utiliza para leer una línea de caracteres de la entrada estándar (sin incluir el cambio de línea final); `printline(s:string)`, que imprime el valor del parámetro `s` a la salida estándar seguido de un cambio de línea del tipo `\r\n`; y `printiline(i:int)`, que imprime el valor del parámetro `i` a la salida estándar seguido de un cambio de línea del tipo `\r\n`.

ACLARACIÓN/MODIFICACIÓN 2

No se permiten declaraciones de tipos con los nombres `int` o `string`. Esto elimina confusiones y evita pérdida de las funcionalidades básicas del lenguaje.

En consecuencia, el siguiente programa no es válido.

```
/* Este programa no es válido. Si lo fuera, a una variable que se declare con
tipo string justo tras la línea A en la sección let-in no podría asignársele
un literal de cadena de caracteres. */

let
      type string = int     /* línea A */
in
      /* ... */
end
```

ACLARACIÓN/MODIFICACIÓN 3

No se permiten declaraciones de funciones cuyos nombres coincidan con los nombres de las funciones de la ***Biblioteca Estándar de TIGER***. Esto elimina confusiones y evita pérdida de las funcionalidades básicas del lenguaje.

En consecuencia, el siguiente programa no es válido.

```
/* Este programa no es válido. Si lo fuera, en la sección in-end no se podría
usar ninguna función que permita escribir una cadena a la salida estándar. */

let
      function print() : int = 0
      function printline() : int  = 0
in
      /* ... */
end
```

ACLARACIÓN/MODIFICACIÓN 4

Las variables de tipo `string` y `array` también aceptan el valor `nil`.

Esto significa que el siguiente es un programa válido.

```
let
      var a : string := nil
      type intArray = array of int
      type intArrayAlias = intArray
      var b : intArray := nil
      var c : intArrayAlias := nil
in
      /* ... */
end
```

ACLARACIÓN/MODIFICACIÓN 5

Las variables y funciones declarados directamente en una sección `let-in` comparten un espacio de nombres (no puede haber dos variables con el mismo nombre, ni dos funciones con el mismo nombre, ni una variable y una función con el mismo nombre), en tanto que los tipos tienen uno diferente que les es propio (no puede haber dos tipos con el mismo nombre).

Por tanto, el siguiente programa no es válido.

```
/* Este programa no es válido. Obsérvese que en la sección let-in aparecen
declarados directamente dos tipos con el mismo nombre (T), y una variable y
una función con el mismo nombre (f). Note, sin embargo, que el hecho de que
existan un tipo y una variable con el mismo nombre (T) no es problema. */

let
      type T = { field : int }
      var f : string := "string"
      type T = { field : string }
      var a : string := nil
      function f(x : int) : int = x + 1
      var T : int := 33
in
      /* ... */
end
```

ACLARACIÓN/MODIFICACIÓN 6

En la expresión de inicialización de arreglos, `type-id [expr1] of expr2`, `expr2` se evalúa `expr1` veces. Esto garantiza que cada una de las posiciones del arreglo tendrá asignada una copia diferente del valor retornado por `expr2` lo cual es significativo cuando, por ejemplo, `expr2` retorna un valor de tipo record o tiene efectos secundarios. Cuando `expr1` retorne `0` no se evaluará `expr2`. Cuando `expr1` retorne un valor negativo se producirá un error en tiempo de ejecución.

El siguiente es un programa válido.

```
/* Este programa es válido. Este programa imprime los números del 1 al 20. */

let
      type ComplexArray = array of SimpleArray
      type SimpleArray = array of int
      var val := 0
      var x := ComplexArray[4] of SimpleArray[5] of (val := val + 1 ; val)
in
      for i := 0 to 3 do
            let
                  var temp := x[i]
            in
                  for j := 0 to 4 do (printi(temp[j]) ; printline(""))
            end
end
```

ACLARACIÓN/MODIFICACIÓN 7

En la expresión de inicialización de records, los campos deben inicializarse en el mismo orden en que aparecen declarados en la definición del tipo record correspondiente.

El siguiente no es un programa válido.

```
/* Este programa no es válido. En la inicialización de los records de tipo
Person, el campo Name debe inicializarse primero que el campo Age. */

let
      type Person = { Name : string, Age : int }
      var john : Person := Person { Age = 22, Name = "John" }
in
      /* ... */
end
```

ACLARACIÓN/MODIFICACIÓN 8

La especificación del lenguaje **TIGER** dice lo siguiente:

"A sequence of zero or more expressions in parenthesis (e.g., (a:=3; b:=a)) separated by semicolons are evaluated in order and returns the value produced by the final expression, if any."

Sin embargo, si alguna de las expresiones que componen una secuencia de expresiones puede *potencialmente* causar un `break` y abortar así la secuencia de expresiones, se considerará que dicha secuencia de expresiones no retorna valor, independientemente del tipo de la última expresión en la secuencia.

El siguiente programa no es válido.

```
/* Este programa no es válido. El compilador debe reportar que la expresión
que se intenta asignar a x en la línea A no retorna valor. */

let
      var x : int := 4
in
      while x > 0 do
            x := (10 ; break ; x - 1)    /* línea A */
end
```

Este fenómeno puede presentarse en escenarios más complejos, como ocurre en el siguiente programa, que tampoco es válido.

```
/* Este programa no es válido. El compilador debe reportar que la expresión
que se intenta asignar a x en la línea A no retorna valor [pues la condición
de la expresión if-then puede potencialmente tomar valor 1 (aunque en este
caso particular eso no ocurra) y hacer que se produzca un break cuando se
ejecute el cuerpo de la misma]. */

let
      var x : int := 4
      var c : int := 8
in
      while x > 0 do
            x := (10 ; if c < 3 then break ; x - 1)    /* línea A */
end
```

Con el objetivo de simplificar el compilador, Ud. puede considerar que el programa siguiente no es válido aunque, a diferencia del anterior, todas las expresiones que intervienen en la condición de la expresión `if-then` son constantes y, por tanto, evaluables en tiempo de compilación.

```
/* A pesar de que el cuerpo de la expresión if-then nunca se ejecutará
(porque su condición siempre evaluará a 0), Ud. puede considerar que este
programa no es válido. El compilador reportará que la expresión que se
intenta asignar a x en la línea A no retorna valor [creyendo que la condición
de la expresión if-then puede potencialmente tomar valor 1 (aunque sabemos
que esto no ocurrirá y que podría haberse detectado en tiempo de compilación)
y hacer que se produzca un break cuando se ejecute el cuerpo de la misma]. */

let
      var x : int := 4
in
      while x > 0 do
            x := (10 ; if 1 - 1 then break ; x - 1)    /* línea A */
end
```

Note que el siguiente programa sí es válido.

```
/* Este programa es válido e imprime la cadena OK. */

let
      var x : int := 1
      var c : int := 10
in
      x := (10 ; while c > 0 do (print("O") ; break) ; x - 1) ;
      print("K")
end
```

ACLARACIÓN/MODIFICACIÓN 9

Se prohíbe la circularidad en la definición de los tipos que introducen arreglos y alias.

El siguiente es un programa inválido.

```
/* Este programa no es válido. MyArray1 está definido circularmente. */

let
      type MyArray1 = array of MyArray2
      type MyArray2 = array of MyArray1Alias
      type MyArray1Alias = MyArray1
in
      /* ... */
end
```

ACLARACIÓN/MODIFICACIÓN 10

Toda expresión *expr* en **TIGER** debe tener un tipo *T* tal que *T* sea "visible" en el contexto en que *expr* aparece. Si *expr* es la expresión que define a todo el programa, su tipo puede ser únicamente `int`, `string`, o el tipo que su implementación asocie internamente a las expresiones que no retornan valor (como el `for`) y a la constante `nil`.

El siguiente no es un programa válido.

```
/* Este programa no es válido. La expresión let-in-end anidada es incorrecta,
pues su tipo (Person) no es "visible" en el contexto en que ella aparece.
(Ese tipo Person no es "visible" en la sección in-end del let exterior.) */

let
      var name : string := "John"
      var age : int := 22
in
      name := "Jane" ;
      let
            type Person = { Name : string, Age : int }
      in
            Person { Name = name, Age = age }
      end ;
      age
end
```

El siguiente programa, sin embargo, sí es válido.

```
/* Este programa es válido. La expresión let-in-end anidada es correcta, pues
aunque es de tipo Entero, su tipo "real" es int (porque Entero es un alias a
int), y el tipo int es "visible" en el contexto en que ella aparece. (El tipo
int es "visible" en la sección in-end del let exterior.) */

let
      var age : int := 22
in
      let
            type Integer = int
            type Entero = Integer
            var edad : Entero := age
      in
            3;
            printline("call");
            "string";
            nil;
            edad
      end
end
```

Tampoco es válido el siguiente código.

```
/* Este programa no es válido. El tipo (Person) de la expresión let-in-end,
que es la que define al programa, no está entre los tipos permitidos para
ella. */

let
      type Person = { Name : string, Age : int }
in
      Person { Name = "Jane", Age = 22 }
end
```

Pero el que sigue, sí es válido.

```
/* Este programa es válido. La expresión let-in-end, que es la que define al
programa, es correcta, pues aunque es de tipo Entero, su tipo "real" es int
(porque Entero es un alias a int), y el tipo int está en la lista de los
permisibles para estas expresiones. */

let
      type Entero = int
      var edad : Entero := 22
in
      edad
end
```

ACLARACIÓN/MODIFICACIÓN 11

Note que el hecho de que la especificación de **TIGER** plantee que…

"In `let` *…* *function-declaration* *…* `in` *expr-seq*$_{opt}$ `end`*, the scope of the function declaration begins at the start of the sequence of function declarations to which it belongs (which may be a singleton) and ends at the* `end`*."*

… es lo que permite que …

"A sequence of function declarations … may be mutually recursive."

De manera análoga ocurre con los tipos.

Considere los ejemplos que se muestran a continuación.

```
/* Este programa es válido.
Aquí hay 4 secuencias de declaraciones:
(1) secuencia de declaraciones de funciones (líneas B-D)
(2) secuencia de declaraciones de variables (líneas E-F)
(3) secuencia de declaraciones de funciones (línea G) [una sola función]
(4) secuencia de declaraciones de tipos     (líneas H-J).
Como las funciones f1 y f2 están en la misma secuencia de declaraciones de
funciones (1) [y, por tanto, ambas son "visibles" desde el comienzo de (1)],
desde  el cuerpo de f1 se puede llamar a f2, a pesar de que f2 está declarada
después que f1. Algo análogo ocurre con las declaraciones de tipos en (4).
Este programa imprime 3628800. */

let                                                                    /* A */
      function f1(n : int) : int = if n <= 0 then 1 else n * f2(n-1)   /* B */
      function writei(i : int) = printi(i)                             /* C */
      function f2(n : int) : int = if n <= 0 then 1 else n * f1(n-1)   /* D */
      var x : int := 10                                                /* E */
      var y : int := 0                                                 /* F */
      function g() = writei(f1(x))                                     /* G */
      type Person = { Name : string, Pet : Animal }                    /* H */
      type Animal = { Name : string, Owner : Individual }              /* I */
      type Individual = Person                                         /* J */
in                                                                     /* K */
      g()                                                              /* L */
end                                                                    /* M */
```

```
/* Este programa no es válido.
Aquí hay 6 secuencias de declaraciones (todas con un único elemento).
La función f1 no puede llamar a f2 porque f2 no es "visible" en el cuerpo de
f1.  (Notar  que  f2  está  declarada  después  que  f1  en  otra  secuencia  de
declaraciones de funciones.) La función f2 sí puede llamar a f1, porque f1
está  declarada  antes  que  f2.  El  campo  Pet  de  Person  no  puede  ser  de  tipo
Animal.  (Notar que Animal está declarado después que Person en otra secuencia
de declaraciones de tipos.) La función g sí puede llamar a f1, porque f1 está
declarada antes que g. El campo Owner de Animal sí puede ser de tipo Person,
porque Person está declarado antes que Animal. */

let
      function f1(n : int) : int = if n <= 0 then 1 else n * f2(n - 1)
      var x : int := 10
      function f2(n : int) : int = if n <= 0 then 1 else n * f1(n - 1)
      type Person = { Name : string, Pet : Animal }
      function g() = printi(f1(x))
      type Animal = { Name : string, Owner : Person }
in
      g()
end
```
