PyTiger2C
Anotaciones sobre el código generado

Yasser González Fernández
yglez@uh.cu

Ariel Hernández Amador
gnuaha7@uh.cu

1. Introducción

Nuestro proyecto se propone desarrollar una implementación de un compilador para el lenguaje de programación Tiger que genere código *C*. Posteriormente, el código *C* generado se compilará con un compilador de *C* para generar un ejecutable para la plataforma específica. El código *C* generado por nuestro compilador será conforme al *standard ISO/IEC 9899:1999*, comúnmente conocido como *C99*, lo cual garantiza que pueda ser procesado por cualquier compilador de *C* que implemente dicho *standard*.

Este documento brinda una descripción general de la estructura y las características del código *C* que generará nuestro compilador.

2. Identificadores

Un identificador en *Tiger* es una secuencia de letras, dígitos y *underscores*, comenzando siempre por una letra. Según la descripción anterior, un identificador en *Tiger* es completamente válido en el lenguaje *C*.

En el código *C* generado se tratará de asignar a un identificador válido de *Tiger* otro con el mismo nombre en *C*, siempre que este no coincida con una palabra reservada del propio lenguaje *C* o con otro identificador definido anteriormente. En caso de que el identificador no sea válido, se le añadirán *underscores* al final hasta lograr un identificador válido.

3. Comentarios

Los comentarios en *Tiger* puede aparecer entre cualquier par *tokens* del lenguaje, enmarcándose entre `/*` y `*/`. El código *C* generado por nuestro compilador no reflejará los comentarios del programa *Tiger* original.

4. Declaraciones de tipos

4.1. Tipos predefinidos

El lenguaje *Tiger* cuenta con dos tipos básicos predefinidos: `int` para números enteros y `string` para las cadenas de caracteres.

El código *C* generado por nuestro compilador creará una variable de tipo `int` para cada variable de tipo `int` en el programa *Tiger* de origen.

Por otra parte, a las variables de tipo `string` de un programa *Tiger* se les asociará una estructura llamada `tiger_string` cuya definición se muestra a continuación.

```
struct tiger_string {
    char *data;
    size_t length;
}
```

El campo `data` corresponde a la secuencia de caracteres de la cadena y el campo `length` almacena a la longitud de la misma.

4.2. *Records*

En *Tiger* los *record* son definidos por una lista de sus campos encerrados entre llaves. Cada elemento de esta lista corresponde a la descripción de un campo y tiene la forma `field_name: field_type` donde `field_type` es un identificador definido con anterioridad o de el propio tipo del *record*.

El código *C* generado por nuestro compilador contendrá una estructura para cada *record* definido en el programa *Tiger* de origen. Cada campo de la estructura corresponderá con uno equivalente en el *record*

definido en *Tiger*. En caso de que algún campo tenga un nombre no válido en *C*, se seguirá la misma estrategia de renombramiento que en el caso de los identificadores.

El siguiente ejemplo muestra el código *C* generado para la definición del *record* `people` en *Tiger* y su representación en el lenguaje *C*.

```
type people { name: string, age: int }
```

```
struct people {
    struct tiger_string *name;
    int age;
}
```

El lenguaje *Tiger* permite la declaración de *records* que tengan campos del propio tipo del *record*, es decir, son definidos en función de ellos mismos. Esta característica de *Tiger* no conlleva ninguna complicación adicional al código *C* equivalente, pues las estructuras de *C* también pueden contener campos del mismo tipo de la estructura que se está definiendo.

El siguiente ejemplo muestra el código *C* generado para la definición de *record* correspondiente a un árbol binario.

```
type binary_tree { value: int, left: binary_tree, right: binary_tree }
```

```
struct binary_tree {
    int value;
    struct binary_tree *left;
    struct binary_tree *right;
}
```

4.3. *Arrays*

En el lenguaje *Tiger* es posible declarar *arrays* de cualquier tipo previamente definido. Nuestro código *C* contendrá una estructura semejante a la usada para el manejo del tipo básico `string` que almacenará en `data` un puntero al primer elemento de la secuencia de datos del *array* y en `length` la cantidad de elementos de el mismo.

El siguiente ejemplo ilustra la creación de un *array* en *Tiger* y el código *C* equivalente generado.

```
type integers = array of int
```

```
struct integers {
    int *data;
    size_t length;
}
```

5. Funciones

En *Tiger* existen dos tipos de funciones, las que no tienen valor de retorno, a las cuales se denomina *procedimientos* y las que tienen un valor y tipo de retorno que se denominan propiamente *funciones*. En nuestro código *C* generado para ambas se sigue la misma idea, con la diferencia de que los procedimientos son generados como funciones `void`, por lo que nos referiremos a los procedimientos como otra función cualquiera.

Tanto las funciones como los procedimientos de *Tiger* definen su propio ámbito o *scope* y a su vez tienen acceso a los identificadores y tipos definidos en el ámbito en que fue definido o su *scope* padre.

En nuestro código *C* generado, para cada función se declarará una estructura `scope` seguida de un número creciente, que tendrá campos para todas las variables declaradas en este y una referencia a la estructura correspondiente al ámbito donde fue definida la función.

Los fragmentos de código *Tiger* que no se encuentren en el cuerpo de una función, se tratarán de modo especial, generando su código *C* equivalente como cuerpo de la función `main`. En este caso también se creará una estructura que defina el ámbito correspondiente, con la única diferencia que no tendrá referencia al ámbito padre.

Una función de *Tiger* tendrá como equivalente una función de *C* de igual nombre, cuyo valor de retorno será del tipo correspondiente al de la función de *Tiger* original y `void` en el caso de los procedimientos. Esta función recibirá como primer parámetro la estructura correspondiente al ámbito padre y a continuación los parámetros equivalentes a los que recibe la función de *Tiger* original. En caso de que existan conflictos con el nombre de la función se seguirá la misma estrategia de renombramiento antes expuesta.

A continuación se muestran algunas definiciones de funciones *Tiger* y el código *C* generado para estas.

- Código que no se encuentre en el cuerpo de ninguna función.

```
let
    var a := 5
    var b := 10
    var c := 0
in
    c := a + b
end
```

```
struct scope1 {
    int a;
    int b;
    int c;
};

int main()
{
    struct scope1* scope;

    /* Reservar memoria para la estructura scope. */

    scope->a = 5;
    scope->b = 10;
    scope->c = 0;
    scope->c = scope->a + scope->b;
}
```

- Declaración de una función simple.

```
let
    var c := 0
    function f(a: int, b:int): int =  a + b
in
    c := f(5, 10)
end
```

```
struct scope1 {
    int c;
```

```
};

struct scope2 {
    struct scope1 *parent;
    int a;
    int b;
};

int f(struct scope1 *parent, int a, int b)
{
    struct scope2 *scope;
    int local_var1;

    /* Reservar memoria para la estructura scope. */

    scope->parent = parent;
    scope->a = a;
    scope->b = b;
    local_var1 = scope->a + scope->b;

    return local_var1;
}

int main()
{
    struct scope1 *scope;

    /* Reservar memoria para la estructura scope. */

    scope->c =  0;
    scope->c = f(scope, 5, 10);
}
```

En *Tiger* se permiten funciones recursivas y funciones definidas en el cuerpo de otra función o también llamadas funciones anidadas. Ninguna de estas características constituyen un impedimento para la estrategia de generación de código antes expuesta. El siguiente ejemplo muestra una función anidada y el código *C* equivalente.

```
let
    var c := 0
    function f(v: int): int =
        let
            function g(h: int): int = v + h
        in
            g(5)
        end
in
    c := f(10)
end

struct scope1 {
    int c;
```

```
};

struct scope2 {
    struct scope1 *parent;
    int v;
};

struct scope3 {
    struct scope2 *parent;
    int h;
};

int f(struct scope1 *parent, int v)
{
    struct scope2 *scope;
    int local_var1;

    /* Reservar memoria para la estructura scope. */

    scope->parent = parent;
    scope->v = v;
    local_var1 = g(scope, 5);

    return local_var1;
}

int g(struct scope2 *parent, int h)
{
    struct scope3 *scope;
    int local_var1;

    /* Reservar memoria para la estructura scope. */

    scope->parent = parent;
    scope->h = h;
    local_var1 = scope->parent->v + scope->h;

    return local_var1;
}

int main()
{
    struct scope1 *scope;

    /* Reservar memoria para la estructura scope. */

    scope->c = 0;
    scope->c = f(scope, 10);
}
```

6. Administración de memoria

Es conocido que en el lenguaje C el programador es el encargado de explícitamente liberar la memoria cuando esta no va a ser utilizada en el resto del programa. Debido a la complejidad que esto implica para la generación de código decidimos utilizar un método de recolección de basura (*garbage collector*, en inglés) que realice esta tarea por nosotros.

Hay varias implementaciones de recolectores de basura para el lenguaje C, seleccionamos *Boehm-Weiser garbage collector* ya que no requiere cambios considerables en la estructura del código. Este recolector de basura está implementado utilizando el algoritmo *mark-sweep*.

Para utilizar este recolector de basura se debe utilizar un macro `GC_MALLOC`, en lugar de la llamada a la función `malloc` habitual.

7. Estructura general del archivo *C*

El código *C* generado por nuestro compilador, para cualquier programa *Tiger*, estará contenido en un único archivo *C* y no deberá tener ninguna dependencia externa además de los *headers* de la biblioteca *standard* de *C*. Este comportamiento hace posible que sólo sea necesario copiar el archivo *C* generado hacia otra máquina y compilar este código sin la necesidad de instalar *headers* o bibliotecas relacionadas con nuestro compilador en esta otra máquina.

Para cumplir con este requerimiento, el archivo *C* generado debe incluir las definiciones de las funciones de la biblioteca *standard* de *Tiger* utilizadas en el programa.

De manera general, el archivo de un programa *C* generado como equivalente de un programa en *Tiger* tendrá la siguiente estructura en su código.

1. **Inclusión de los *headers* correspondientes a las funciones de la biblioteca *standard* de *C* utilizados en el programa.**
2. **Declaraciones de tipos.**
   a) **Tipos de la biblioteca *standard* de *Tiger*.** En esta parte del código se encontrarán las declaraciones de los tipos `int` y `string` además de cualquier tipo que sea necesario para la implementación de las funciones de la biblioteca *standard* de *Tiger*.
   b) **Tipos definidos en el programa.** En esta parte del código se encontrarán las declaraciones equivalentes de los tipos definidos en el programa *Tiger* de origen.
3. **Declaración de los *scopes*.** En esta parte del código se encontrarán las declaraciones de los *scopes* asociados a cada función del programa *Tiger* de origen y a la función `main` de *C*.
4. **Prototipos de las funciones.** Al colocar los prototipos en esta parte del código se garantiza que las funciones serán accesible para las que los necesiten sin importar el orden en que estas sean definidas más adelante.
   a) **Prototipos de las funciones de la biblioteca *standard* de *Tiger*.**
   b) **Prototipos de las funciones definidas en el programa *Tiger*.**
5. **Cuerpo de las funciones.**
   - **Cuerpo de las funciones de la biblioteca *standard*.** En esta parte del código se encontrarán las implementaciones de las funciones de la biblioteca *standard* de *Tiger* que se utilicen en el programa *Tiger* en cuestión.
   - **Cuerpo de las funciones definidas en el programa *Tiger*.** En esta parte del código estará el código *C* equivalente a cada función definida en el programa *Tiger* de origen.
6. **Función *main*.** Esta función recibe un trato especial y en su cuerpo se encontrará el código *C* equivalente las instrucciones que no se encuentran dentro de una declaración de función o de tipo.