Compiladores 2021-1
Facultad de Ciencias UNAM
Proyecto Final

Profesora: Lourdes del Carmen González Huesca
Ayudante: Juan Alfonso Garduño Solís
Laboratorio: Fernando Abigail Galicia Mendoza

Nombre del equipo:

VeryBlueBerries

Integrantes del equipo:

- Becerril Torres Teresa
  - No. cuenta: 315045132
  - Correo electrónico: terebece1508@ciencias.unam.mx
- Sánchez Benítez María del Pilar
  - No. cuenta: 315239674
  - Correo electrónico: pilar_sb_cc@ciencias.unam.mx
- Torres Sanchez Miguel Ángel
  - No. cuenta: 315239674
  - Correo electrónico: miguel.atorress@ciencias.unam.mx

Enlace del repositorio:

https://github.com/Terebece/Proyecto-de-VeryBlueBerries

Uso del compilador:

1. Clonar el repositorio utilizando
   $ git clone https://github.com/Terebece/Proyecto-de-VeryBlueBerries.git
2. Ingresar a la carpeta Proyecto-de-VeryBlueBerries, en la cual se encuentran los siguientes archivos:
   a. Front-end.rkt : Archivo con todos los procesos de Front-end.
   b. Middle-end.rkt : Archivo con todos los procesos de Middle-end.
   c. Back-end.rkt : Archivo con todos los procesos de Back-end.
   d. Compiler.rkt : Archivo que compila un archivo con extensión .mt
   e. ejemplo1.mt, ejemplo2.mt y ejemplo3.mt son archivos con con código LF.

3. Para compilar un archivo con extensión .mt se debe abrir el archivo Compiler.rkt y especificar en (define path "ejemplo1") el nombre del archivo, ya sea utilizando los ejemplos que se proponen o creando un archivo nuevo en la carpeta Proyecto-de-VeryBlueBerries.
4. Compilar el archivo Compiler.rkt, mostrará en la consola las fases del compilador y generará los archivos con extensión .fe, .me y .c.

Explicación de cada etapa del proceso de compilación:

- **Front-end:**

  En la fase de front-end de nuestro compilador se revisa que no existan errores de escritura y verificamos que no existan variables libres en el programa ni errores en la aridad de las operaciones primitivas. Para esto realizamos los siguientes procesos para ir transformando el lenguaje fuente en el orden en que aparecen.

  - **remove-one-armed-if**

    Proceso que se encarga de quitar las expresiones *if* de una sola rama del lenguaje. Cambia estas expresiones por *if* de dos ramas donde la rama que corresponde al else está vacía.

  - **remove-string**

    Proceso encargado de eliminar las cadenas de caracteres del lenguaje y las convierte en una lista de caracteres de nuestro lenguaje.

  - **curry-let**

    Proceso que currifica las expresiones *let* y *letrec* con el fin de tener una sola asignación para sus constructores y facilitar la construcción de código en C donde cada *let* es una asignación y los *letrec* una definición de función.

  - **identify-assignments**

    Proceso que identifica si un *let* se utiliza para definir funciones y lo reemplaza por un *letrec*. Así, tenemos todas las definiciones de funciones dentro de un *letrec*.

  - **un-anonymous**

    Proceso encargado de asignarle un nombre a las funciones anónimas *lambda*. Es necesario que todas las funciones tengan nombre para poder generarlas en código C. Se agrega el constructor de asignación *letfun* para las funciones con nombre que recibe un identificador, el tipo *Lambda*, la *lambda* como valor y como cuerpo una llamada a la función.

- **verify-arity**
  Proceso que funciona como un verificador de la sintaxis que consiste en verificar el número de parámetros que reciben las primitivas. Debe corresponder con su respectiva aridad. Para las expresiones aritméticas (+, -, *, /) y para el *and* y *or* su aridad debe ser de al menos 2, para *not, car, cdr y length* su aridad debe ser igual a 1. Regresa la misma expresión si su aridad corresponde, en caso contrario lanza un error.

- **verify-vars**
  Proceso que funciona como verificador de la sintaxis que consiste en verificar que una expresión no tiene variables libres. Regresa un error si encuentra variables libres o la misma expresión si no las hay.

❖ Middle-end:

Para el middle-end nuestro compilador hace inferencia de tipos y agrega anotaciones de tipos a las constantes del lenguaje. Se utiliza el algoritmo J y para facilitar esta tarea se currifican temporalmente las expresiones *lambda* y aplicaciones de función. Al finalizar, se deshace la currificación de las expresiones *lambda.* Los procesos que se realizan se muestran en el orden en el que se ejecutan a continuación.

- **curry**
  Proceso que currifica las expresiones *lambda* y aplicaciones de función para facilitar el cambio de tipo *Lambda* a tipo función.

- **type-const**
  Proceso encargado de agregar las anotaciones de tipo a las constantes del lenguaje y facilitar la inferencia de tipos y generación de código C.

- **type-infer**
  Proceso que quita las anotaciones de tipo *Lambda* y las sustituye por el tipo función *(T →T)* y las anotaciones de tipo *List* por *(List of T).* Se utiliza el algoritmo J para inferir el tipo *T* y hacer las anotaciones correctas.

- **uncurry**
  Proceso que deshace la currificación de las expresiones *lambda* hecha al inicio de la fase del middle-end. *Descurrificar* nos permitirá hacer funciones multiparamétricas al traducir al lenguaje C.

❖ Back-end:

En el back-end nuestro compilador hace la traducción final al lenguaje C. Para esto se hacen dos últimos procesos en el orden siguiente.

- **list-to-array**
  Proceso que cambia las listas al arreglos. Los arreglos reciben una constante que es su longitud, el tipo de sus elementos y una lista de expresiones del mismo tipo que vienen de la lista que se está transformando. Este proceso nos permitirá escribir nuestras listas como arreglos en el lenguaje C.

- **c**
  Es el proceso final que traduce nuestro lenguaje a código C en una cadena de texto.