Universidad Nacional Autónoma de México

Facultad de Ciencias

Compiladores

Compilador: LF a C

Profesora **Lourdes del Carmen González Huesca**

Ayudante Naomi Itzel Reyes Granados

Ayud. Lab. Fernando Abigail Galicia Mendoza

Ayud. Lab. Nora Hilda Hernández Luna

Ángel Christian Pimientel Noriega; 316157995; cristianp@ciencias.unam.mx,

Juan García Lugo; 316161013; juanlugo@ciencias.unam.mx,

Nestor Semer Vazquez Cordero; 316041625; nestor2502@ciencias.unam.mx

29 de enero del 2022

1. Ejecutar

Para compilar un archivo con extensión .mt se debe abrir el archivo Compiler.rkt y especificar en (define path) el nombre del archivo, ya sea utilizando los ejemplos que se proponen o creando un archivo nuevo en la carpeta examples.

2. Etapas de compilación

1. **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, y para poder manejar mejor las expresiones se currifican.

   a) **remove-one-armed-if** (práctica 3)
   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.

   b) **remove-string** (práctica 3)
   Proceso encargado de eliminar las cadenas de caracteres del lenguaje y las convierte en una lista de caracteres de nuestro lenguaje.

   c) **curry-let** (práctica 4)
   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.

   d) **identify-assigments** (práctica 4)
   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*

   e) **un-anonymous** (práctica 4)
   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.

   f) **verify-arity** (práctica 4)
   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.

   g) **verify-vars** (práctica 4)
   Proceso 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.

h) **curry** (práctica 5)
Proceso que currifica las expresiones *lambda* y aplicaciones de función para facilitar el cambio de tipo *Lambda* a tipo función.

2. **Middle-end** (práctica 5)
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**. 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.

   a) **type-const** (práctica 5)
   Proceso encargado de agregar las anotaciones de tipo a las constantes del lenguaje y facilitar la inferencia de tipos y generación de código C.

   b) **type-infer** (práctica 5)
   Proceso que quita las anotaciones de tipo *Lambda* y las sustituye por el tipo función $(T \to T)$ 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.

   c) **uncurry** (práctica 6)
   Proceso que deshace la currificación de las expresiones lambda hecha al final de la fase front-end. Descurrificar nos permitirá hacer funciones multiparamétricas al traducir al lenguaje C.

3. **Back-end**
En el back-end nuestro compilador hace la traducción final al lenguaje C. Para esto se hacen tres procesos en el orden siguiente.

   - **list-to-array** (práctica 6)
   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.
   - **assigment** (práctica 6)
   Modifica los constructores *let*, *letrec* y *letfun*, eliminando el valor asociado a los identificadores y el tipo correspondiente, y asigna en la tabla de símbolos el valor de las variables
   - **c**
   Es el proceso final que traduce nuestro lenguaje a código C en una cadena de texto.